MODELS AND TECHNIQUES FOR THE VISUALIZATION OF LABELED DISCRETE OBJECTS

By

DINESH P. MEHTA

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

1992

UNIVERSITY OF FLORIDA LIBRARIES

ACKNOWLEDGEMENTS

I wish to express my gratitude to my advisor, Professor Sartaj Sahni, for his guidance, encouragement, and help over the last five years. This dissertation would not have been possible without his supervision. I am also grateful to the members of my dissertation committee at Florida, Professors Gerhard Ritter, John Staudhammer, Ted Johnson, and Haniph Latchman for patiently reading this manuscript and providing insightful feedback. I thank Phil Barry of the Computer Science Department and Bill Fox of the Department of Psychology at the University of Minnesota for their support. I also thank the Computer Science Departments at the University of Florida and the University of Minnesota, and the National Science Foundation for financial support.

I would also like to thank my friends, Mario Lopez, Vijay Rajan, Rose Tsang, and Andrew Lim for many fruitful discussions and also for other not-so-fruitful, but entertaining, times which made the process enjoyable.

Finally, I wish to thank my family, and in particular, my parents, Prabha and Prakash Mehta, for their love and support. This dissertation is dedicated to them.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS ii

ABSTRACT v

CHAPTERS

1 INTRODUCTION 1

2 STRING VISUALIZATION 3

2.1 Problem Specification 3
2.2 Visualization Conflicts 5
2.3 Refinements of Display Model 6
2.4 String Visualization Queries 9
2.5 Applications 10
2.5.1 General Methods 10
2.5.2 Numerical Data 12
2.5.3 Molecular Biology 13
2.5.4 Textual Data 14

3 STRING VISUALIZATION ALGORITHMS 15

3.1 Definitions 15
3.1.1 Compact Symmetric Directed Acyclic Word Graphs (csdawgs) 15
3.1.2 Computing Occurrences of Displayable Entities in a string . . 17
3.1.3 Prefix and Suffix Extension Trees 17
3.2 Computing Conflicts 18
3.2.1 Algorithm to determine whether a string is conflict-free 18
3.2.2 Subword Conflicts 21
3.2.3 Prefix-Suffix Conflicts 26
3.2.4 Alternative Algorithms 39
3.3 Size Restricted Queries 42
3.4 Pattern Oriented Queries 43
3.5 Statistical Queries 45
3.6 Experimental Results 47
3.7 Display Algorithms 51

4 CIRCULAR STRING VISUALIZATION 60

4.1 Introduction 60
4.2 Definitions 62
4.3 Constructing the Csdawg for a Circular String 64
4.4 Computing Conflicts Efficiently 74
4.5 Applications 75

5 EXTENSION TO BINARY TREES AND SERIES-PARALLEL GRAPHS 80

5.1 Tree Visualization 80
5.1.1 Problem Definition 80
5.1.2 Applications 81
5.1.3 Algorithms 85
5.2 Geometric Series-Parallel Graph Visualization 88
5.2.1 Problem Definition 88
5.2.2 Applications 91
5.2.3 Algorithms 91

6 SYSTEM INTEGRATION 94

6.1 Using the Object-Oriented Methodology 94
6.2 Overview of a visualization system 95

7 CONCLUSIONS 100

REFERENCES 103

BIOGRAPHICAL SKETCH 105

Abstract of Dissertation
Presented to the Graduate School of the University of Florida
in Partial Fulfillment of the Requirements for the
Degree of Doctor of Philosophy

MODELS AND TECHNIQUES FOR THE VISUALIZATION OF LABELED DISCRETE OBJECTS

By

Dinesh P. Mehta

August 1992

Chairman: Dr. Sartaj Sahni
Major Department: Computer and Information Sciences

The objective of visualization is to extract useful and relevant information from raw data and present it so that it can be easily understood and assimilated by humans.

We propose a general model for the visualization of labeled discrete objects. (Examples of labeled discrete objects are strings, circular strings, trees, graphs, etc.) Our model is based on identifying similar subobjects in an object and coding them visually. This is demonstrated by applying the model to linear and circular strings, binary trees, and series-parallel graphs. We also describe and classify the problem of display conflicts that is associated with this visualization model and suggest methods to overcome it. We extend the csdawg data structure for linear strings to circular strings. Efficient, often optimal, algorithms to implement the visualization model for linear and circular strings, binary trees, and series-parallel graphs are also developed. These algorithms are specifically designed to utilize the inheritance and data abstraction features of the object-oriented paradigm. Several of these algorithms were implemented in C++ to evaluate their performance. We also propose a blueprint for a visualization system based on our visualization model.

Applications of this visualization technique arise in areas such as molecular biology, computer vision, computer graphics, CAD VLSI, data compression, algorithm animation, and debuggers.

CHAPTER 1
INTRODUCTION

The objective of visualization is to extract useful and relevant information from data and represent it so that it can be easily understood and assimilated by humans. This enables specialists in application areas to observe trends and patterns in the data. This could lead to better understanding of various phenomena and provide insights resulting in theories or hypotheses, which can subsequently be proved (or disproved) by formal methods or by further experimentation.

A notion that is useful in the understanding of objects is that of *similarity.* Multiple occurrences of the same pattern in data which represents the outcome or result of some process indicates the presence of many instances of the same "effect." Analyzing the set of circumstances associated with these occurrences could yield a plausible "cause." For example, multiple occurrences of the same flaw in a paper roll might reveal the faulty component in the paper production process.

Similarly, multiple occurrences of the same patterns in an object whose structure is being studied indicates the presence of many instances of the same "cause." Observing the phenomena which occur in the presence of these patterns could shed some light on the "effect" of the patterns. For example, the presence of multiple occurrences of patterns in DNA strands in organisms could manifest themselves as common characteristics shared by the organisms.

This linkage of cause and effect is a fundamental goal in many scientific disciplines. Most work in visualization so far, which attempts to facilitate the scientific goals outlined above, consists of choosing methods to display *individual units of data*, so

that patterns and trends become visually obvious when the data is seen in its entirety [1, 2]. However, the onus of detecting patterns and trends still lies on the user or the specialist. This becomes more crucial when the amount of data is large and the user's perceptual faculties are overburdened. Consequently, errors of omission (not seeing patterns which are actually there) become more likely.

Our work attempts to shift the responsibility of detecting patterns from the human to the computer. This is done by devising algorithms to detect patterns in the data and then making these patterns available to the user for further scrutiny. Multiple occurrences of the same pattern can be made visually explicit by color coding them with the same color. Other methods could also be used, such as "flashing" occurrences of the same pattern on the screen. If visual schemes are not appropriate, recurring patterns could simply be provided as a list of occurrences which the user goes through. In the context of a visualization environment, this technique could be used either in a stand-alone manner or as a supplement to other visualization methods.

For the technique outlined above to be useful, we propose the following principles on displaying patterns, which we shall call the *Similarity Paradigm.* The similarity paradigm forms the basis for the visualization techniques used in this dissertation.
Principle 1: Two patterns should be displayed to *look* similar iff they *are* similar.
Principle 2: The *degree of similarity* between the displays of two patterns should be proportional to the *actual similarity* of the patterns.

Chapter 2 discusses string visualization while Chapter 3 provides algorithms which implement string visualization queries. Chapter 4 discusses the extension of our visualization techniques for linear strings to circular strings and Chapter 5 deals with the visualization of binary trees and series-parallel graphs. Finally, Chapter 6 explains how the work of the first five chapters may be integrated to form a visualization system based on the similarity paradigm.

CHAPTER 2
STRING VISUALIZATION

2.1 Problem Specification

A complete specification of a visualization problem based on the similarity paradigm requires one to provide the following five items. These are illustrated using linear string visualization as an example. Henceforth, the word "string" shall refer to linear strings.

(1) Structure of the Data to be Visualized: Does the data represent a string, a series-parallel graph, a binary tree, etc?

In the string visualization example, the data is a string S of length n whose characters are chosen from a fixed alphabet Σ of constant size.

(2) Structure of Patterns: This depends largely on the structure of the data. If the data represents a string, then the structure of the pattern could be a substring (contiguous sequence of string elements), a subsequence (a non-contiguous sequence of string elements), etc. Patterns can have other constraints imposed upon them. For example, a pattern may be required to be of a minimum size.

In the string visualization example, the pattern is a substring of S defined uniquely by its start and end positions.

(3) Maximality of Patterns: If a pattern is repeated in the data, then any subpattern of the pattern is also repeated. For example, if abc repeats in a string, so do ab, bc, a, b, and c . In particular, if all occurrences of ab occur in the context of abc, then attempting to distinguish between ab and abc does not serve any useful purpose.

Defining maximality and restricting the user's attention to maximal patterns helps to simplify the display.

In the string visualization example, a pattern is said to be *maximal* iff its occurrences are not all preceded by the same letter, nor all followed by the same letter. Consider the string $S = abczdefydefxabc$. Here, the empty string λ, abc, def, and S are the only maximal patterns. The occurrences of def are preceded by different letters (z and y) and followed by different letters (y and x). The occurrences of abc are not preceded by the same letter (the first occurrence does not have a predecessor) nor followed by the same letter. However, de is not maximal because all its occurrences in S are followed by f.

(4) Measure of Similarity (MS): A measure of similarity would consist of attaching numerical values to pairs of maximal patterns which indicate the degree of similarity between the two patterns.

In the string visualization example, the measure of similarity could be defined as follows: If two patterns are identical, then **MS** = 1. Otherwise, **MS** = 0. In other words, two patterns are defined to be similar iff they are identical. There is no concept of "degree of similarity" in this definition.

(5) Display Models: This addresses the issues of which patterns are displayed and how they are displayed. While choosing display models, Principles 1 and 2 of Chapter 1 should be kept in mind.

In the string visualization example, a pattern is said to be a *displayable entity* (or displayable) iff it is maximal, non-null, and occurs more than once in S (in this case, all maximal patterns are displayable entities with the exception of S, which occurs once in itself and λ) . All instances of the same displayable entity are highlighted in the same color. Instances of different displayable entities are highlighted in different

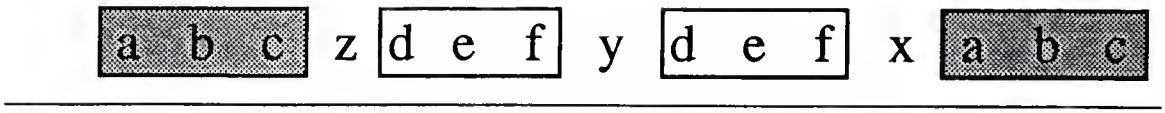


Figure 2.1. Highlighting Displayable Entities

colors (there is no relationship between colors representing different displayable entities). In the example string S, *abc* and *def* are the only displayable entities. So, S would be displayed by highlighting *abc* in one color and *def* in another as shown in Figure 2.1.

2.2 Visualization Conflicts

Consider the string S = *abcicdefcdegabchabcde* and its displayable entities, *abc* and *cde* (both are maximal and occur thrice). The displayable entities *abc* and *cde* must be highlighted in different colors. Notice, however, that *abc* and *cde* both occur in the string *abcde*, which is a suffix of S. Clearly, both displayable entities cannot be highlighted in different colors in *abcde* as required by the model. This is a consequence of the fact that the letter *c* occurs in both displayable entities. This situation is known as a prefix-suffix conflict because a prefix of one displayable entity (here, *cde*) is a suffix of the other (here, *abc*). Note, also, that *c* is a displayable entity in S. Consequently, all occurrences of *c* must be highlighted in a color different from those used for *abc* and *cde*. But this is impossible as *c* is a subword of both *abc* and *cde*. This situation is referred to as a subword conflict. Formally,

(i) A *subword conflict* between two displayable entities D_1 and D_2 in S exists iff D_1 is a substring of D_2.

(ii) A *prefix-suffix conflict* between two displayable entities D_1 and D_2 in S exists iff there exist non-null substrings, S_p, S_m, S_s in S such that $S_pS_mS_s$ occurs in S, S_pS_m

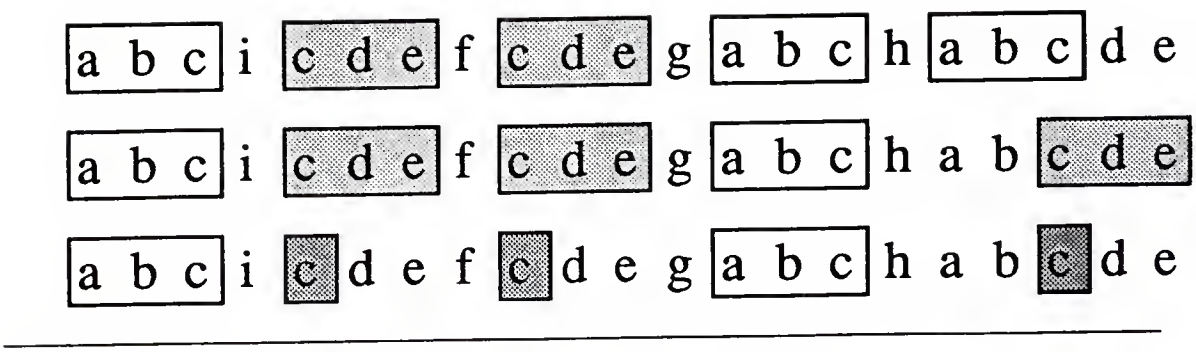


Figure 2.2. Possible Configurations

$= D_1$, and $S_m S_s = D_2$. We say that a prefix-suffix conflict exists between D_1 and D_2 *with respect to* S_m. S_m is known as the *intersection* of the conflict.

2.3 Refinements of Display Model

When subword and prefix-suffix conflicts occur, we need some criteria to determine which of the information previously required to be displayed actually gets displayed. For instance, in the example string S = *abcicdefcdegabchabcde* from the previous section, three possible non-conflicting, displayable subsets are shown in Figure 2.2. In this section we present three refinements to the display model from Section 2.1 which attempt to overcome the display difficulties created by conflicts. They are

(1) One-Copy, Maximum-Content, No-Overlap: In this model, exactly one copy of the string is displayed. Occurrences of displayable entities are selected so that there are no mutually conflicting occurrences. Given this restriction, the model requires occurrences to be selected so that the amount of information conveyed by the display is maximized. This goal may be achieved in three ways.

Interactive : The user selects occurrences interactively by using his/her judgement. Typically, this would be done by examining the occurrences which are involved in a conflict and choosing one that is the most meaningful.

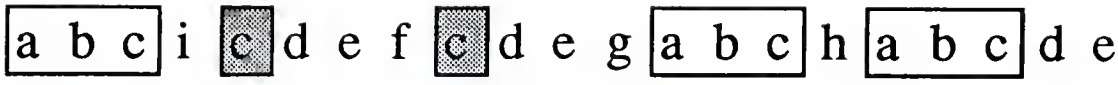


Figure 2.3. Optimal Configuration under Model 1

Automatic : A numeric weight is assigned to each occurrence. The higher the weight, the greater the desirability of displaying the corresponding occurrence. Criteria that could be used in assigning weights to occurrences include length, position, number of occurrences of the pattern, semantic value of the displayable entity, information on conflicts, etc. The information is then fed to a routine which selects a set of occurrences so that the sum of their weights is maximized. For example, consider string $S = abcicdefcdegabchabcde$ of Section 2.2. If the weight assigned to each occurrence of *abc* is 4, *cde* is 2, *c* is 3, then Figure 2.3 shows the optimal display configuration. The total weight of the display is 18.

Semi-Automatic: In a practical environment, the most appropriate method would be a hybrid of the Interactive and Automatic approaches described above. The user could select some occurrences that he/she wants included in the final display. The selection of the remaining occurrences can then be performed by a routine which maximizes the display information.

(2) Multiple-Copy, No-Overlap: Multiple copies of the string may be displayed. Mutually disjoint sets of occurrences are associated with the copies (one set per copy), so that the occurrences corresponding to each copy are mutually non-conflicting. There are two approaches:

(a) A constant number (max) of copies of the string may be displayed. The total content of the display, summed over all max copies, is to be maximized. For example,

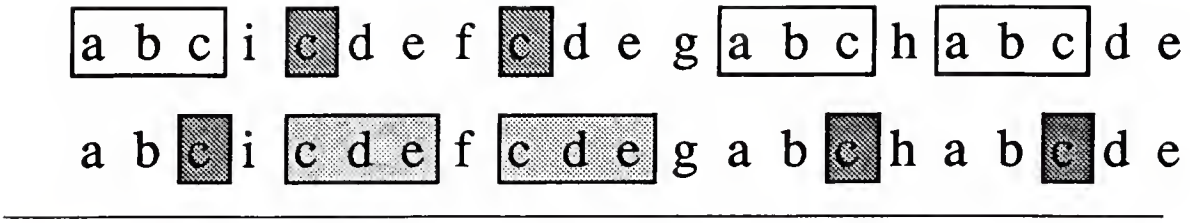


Figure 2.4. Optimal Configuration under Model 2(a)

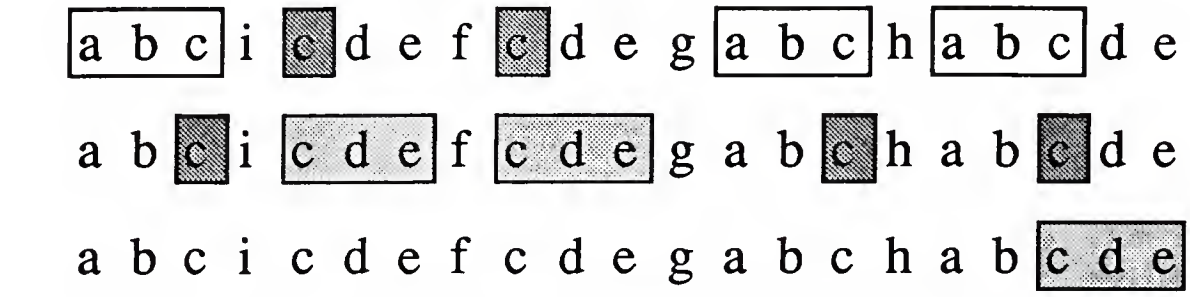


Figure 2.5. Configuration under Model 2(b)

consider string $S = abcicdefcdegabchabcde$ of Section 2.2. If the weights corresponding to *abc*, *cde*, and *c* are 4, 2, and 3, respectively, and $max = 2$, then Figure 2.4 shows the optimal display configuration. The total weight of the display is 31.

(b) No limit is imposed on the number of copies of the string that may be displayed. Each occurrence is highlighted in exactly one copy of the string. The number of copies of the string used should be minimized. In the worst case, $O(n^2)$ copies of the string may be required. The displayable entities in the string $S = a^n$ are a^i, for $1 \leq i < n$. So, each proper substring of a^n is an occurrence of a displayable entity of S. Consider the character $S_{n/2}$. It occurs in approximately $n^2/4$ occurrences of displayable entities (since there are approximately $n/2$ positions to the left and to the right of $S_{n/2}$ and all but one combination of these represents an occurrence of a

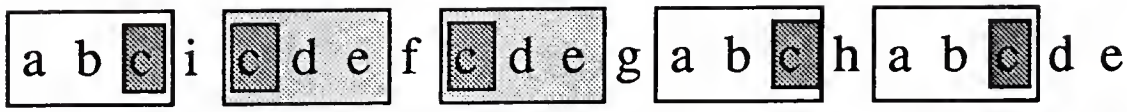


Figure 2.6. Optimal Configuration under Model 3

displayable entity). Since a different copy of the string is required to display these occurrences without overlap, the number of copies of the string is approximately $n^2/4$ or $O(n^2)$. Figure 2.5 shows a configuration for string S = *abcicdefcdegabchabcde* of Section 2.2.

(3) Single-Copy, Maximum-Content, Subword-Overlap: Exactly one copy of the string may be displayed. Occurrences are selected so that no pair of occurrences has a prefix-suffix conflict. Subword conflicts are allowed. As in the Single-Copy, Maximum-Content, No-Overlap model, the goal is to maximize the information conveyed. Again, there are three approaches for selecting occurrences: Automatic, Interactive and Semi-Automatic. For example, consider string S = *abcicdefcdegabchabcde* of Section 2.2. If the weights corresponding to each occurrence of *abc*, *cde*, and c are 4, 3, and 2, respectively, then Figure 2.6 shows the optimal display configuration. The total value of the display is 31.

Note that it is crucial to all methods to get information on prefix-suffix and subword conflicts.

2.4 String Visualization Queries

The following algorithmic problems arise as a result of the discussion in the previous section:

Given a string S of length n whose elements are chosen from an alphabet Σ of fixed size,

P1. Obtain a list of all displayable entities and their occurrences.

P2. Obtain a list of all prefix-suffix conflicts.

P3. Obtain a list of all subword conflicts.

Given a list of occurrences of displayable entities and a weight associated with each occurrence,

P4. Obtain a set of mutually non-conflicting occurrences so that the sum of the weights associated with them is maximum (Model 1).

P5. Obtain *max* mutually disjoint sets of mutually non-conflicting occurrences so that the sum of the weights associated with them is maximum (Model 2(a)).

P6. Obtain a minimum number of mutually disjoint sets of mutually non-conflicting occurrences required to partition the set of all occurrences (Model 2(b)).

P7. Obtain a set of occurrences, such that no two occurrences in the set have a prefix-suffix conflict, so that the sum of the weights associated with them is maximized (Model 3).

Algorithms for **P1**-**P4**, **P6**, and **P7** are presented in Chapter 3.

2.5 Applications

An abstract strategy has been outlined for the visualization of strings. This section discusses some general methods for applying this strategy to specific areas. Applications to molecular biology, text and numerical sequences are also outlined.

2.5.1 General Methods

In order to apply this strategy successfully to actual data, it is important to first check that the data conforms to the definition of strings provided in Section 2.1. If this is not the case, then it may be possible to transform the data so that it satisfies the definition without losing vital information in the process.

(1) If a string consists of characters which are chosen from a fixed alphabet of a small size (≤ 50, say), then it is already in the correct format. For example, DNA sequences are made up from an alphabet of 4 elements. English text is made up of an alphabet of 26 characters and some special symbols.

(2) Otherwise, if a function, f, can be defined for each element in the alphabet such that:

a) The range of f can be determined and is of constant size and

b) $f(element_1) = f(element_2)$ iff $element_1$ and $element_2$ are similar,

then the given string may be converted to another string which is obtained by applying f to each element of the original string. The resulting string can now be input to the visualization routines.

For example, consider a sequence of objects which are chosen from a large (possibly infinite) alphabet. Assume that a set of properties, $P = \{P_1, P_2, \ldots, P_m\}$, is associated with each object. Suppose that patterns of property P_i of the sequence of objects are interesting (where P_i may take on one of a constant, fixed number of values). Then, for the purposes of visualization, each object in the sequence is replaced by the corresponding P_i value. This approach can be used with the other properties as well. Some examples where this approach is useful are

(1) *Protein Sequences* [2]: A protein sequence consists of a sequence of amino acids. While the number of amino acids that could form a sequence is large , it is possible to place amino acids in groups on the basis of physical properties such as hydrophobicity, acidity, polarity etc. Amino acids in the sequence may then be replaced by a symbol representing the groups to which they belong.

(2) *Chronological sequences of Multidimensional Data* [1]: Here, a number of measurements relating to a particular scientific phenomenon are taken at regular intervals of time. The measurements for each variable are classified as *low, medium*, or *high.*

Consequently, the sequence of multidimensional data may be replaced by a sequence of symbols L, M, H representing low, medium, and high values, respectively, of the variable being studied.

Finally, many applications would benefit by comparisons between two or more *different strings* (as opposed to comparisons within the same string). This can also be supported by a simple extension of our techniques.

2.5.2 Numerical Data

An important category of data is numerical data. These arise whenever properties of objects are described by measurements. Numerical information, in general, is chosen from large alphabet sizes which are determined by the accuracy of measurement required. Clearly, numerical sequences cannot be directly input to our visualization system. This is remedied by determining the range of values that a variable can take on. This range is then subdivided into a constant number of subranges (this is essentially the same strategy used by Beddow [1]). Each value in the sequence is then replaced by a symbol representing the subrange to which it belongs. The resulting sequence may then be input to our visualization system. Consider a sequence of numbers which lie in the range 1-200. Assume that subranges have been defined as 1-20, 21-40, ..., 181-200 which are represented by the symbols a, b, ..., j, respectively. So, the sequence 7 142 63 94 6 148 69 becomes *ahd* e *ahd*.

Sequences of numbers, such as financial data, are usually studied by using graphs. We expect that the techniques outlined here if used in conjunction with graphs could reduce the possibility of overlooking important patterns in the data. This can be done by either appropriately coloring pieces of the graph or by coloring a string which is aligned with the graph.

Often, comparisons are made not between the values of numbers in a sequence, but between the increase/decrease in consecutive values. For example, in [5 20 15 75 90 85], 5 20 15 is not obviously related to 75 90 85. However, the increases/decreases in values are identical. An increase of 15 followed by a decrease of 5. Information of this type may be obtained by transforming the string by taking the difference between successive values before inputting it to the visualization system; i.e., 15 -5 60 15 -5. Similar transformations may be used for percentage increases/decreases, second order differences, etc.

2.5.3 Molecular Biology

In molecular biology, RNA, DNA, and protein sequences are studied. Sequence comparison helps to answer questions about evolution, structure and function in organisms, and the structural configuration of individual RNA molecules [3]. Of particular importance are repeating patterns and their relative positions [4]. Here [4], the csdawg data structure, which is described in the next chapter, is used to analyze sequences. Our work improves upon this by suggesting more effective display methods and by introducing more sophisticated analysis techniques involving prefix-suffix and subword conflicts.

For example, let D_1 and D_2 be displayable entities and D_1 be a subword of D_2. If the fraction (*number of occurrences of* D_1 *contained in* D_2)/(*number of occurrences of* D_1) ≈ 1, then we can infer that D_1 usually occurs as a subword of D_2 which could mean that D_1 does not perform any significant functions except as a subword of D_2.

Suppose patterns P_1 and P_2 perform functions F_1 and F_2 in an organism. Then, if (*number of prefix-suffix conflicts between* P_1 *and* P_2)/ (*min*{number of occurrences of P_1,number of occurrences of P_2}) ≈ 1, then we can infer that F_1 and F_2 are generally performed by the same region and are therefore related in some way.

2.5.4 Textual Data

Structural information about text may be obtained by studying prefix-suffix and subword conflicts. Information about the *contexts* in which certain phrases are used is provided by subword conflicts. Information on the *combination of phrases* is provided by prefix-suffix conflicts. This information can be used to identify anomalies in sentence structure and possibly identify the author of a text by the structure. It can also be used to decipher text coded using sophisticated substitution ciphers where patterns are substituted by other patterns.

CHAPTER 3
STRING VISUALIZATION ALGORITHMS

3.1 Definitions

3.1.1 Compact Symmetric Directed Acyclic Word Graphs (csdawgs)

The csdawg data structure data structure is used to represent a string or a set of strings. It evolved from other string data structures such as position trees [5], suffix trees [6], directed acyclic word graphs [7, 8], etc. A csdawg $CSD(S)$ corresponding to a string S is a directed acyclic graph defined by a set of vertices $V(S)$, a set $R(S)$ of labeled directed edges called right extension edges (re-edges), and a set $L(S)$ of labeled directed edges called left extension edges (le-edges). Each vertex of $V(S)$ represents a maximal substring of S. Specifically, $V(S)$ consists of a source (which represents the empty word, λ), a sink (which represents S), and a vertex corresponding to each displayable entity of S.

Let $str(v)$ denote the string represented by vertex v, for $v \epsilon V(S)$. Define the *implication*, $imp(S, \alpha)$, of a string α in S to be the smallest superword of α in $\{str(v)$: $v \epsilon V(S)\}$, if such a superword exists. Otherwise, $imp(S, \alpha)$ does not exist.

Re-edges from a vertex v_1 are obtained as follows: for each letter x in Σ, if $imp(S, str(v_1)x)$ exists and is equal to $str(v_2) = \beta str(v_1)x\gamma$, then there exists an re-edge from v_1 to v_2 with label $x\gamma$. If β is the empty string, then the edge is known as a *prefix extension edge*. Le-edges from a vertex v_1 are obtained as follows: for each letter x in Σ, if $imp(S, xstr(v_1))$ exists and is equal to $str(v_2) = \gamma x str(v_1)\beta$, then

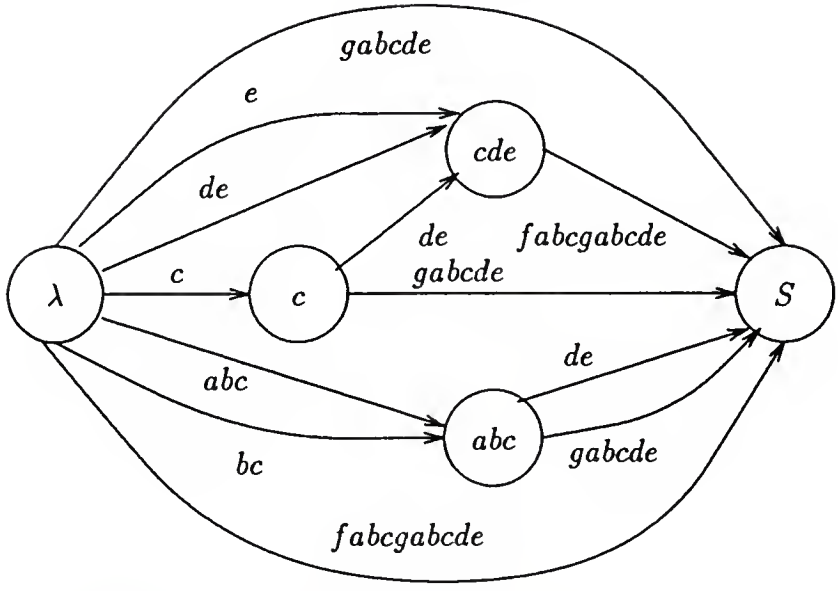


Figure 3.1. Csdawg for $S = cdefabcgabcde$ ($L(S)$ not shown)

there exists an le-edge from v_1 to v_2 with label γx. If β is the empty string, then the edge is known as a *suffix extension edge.*

Figure 3.1 shows $V(S)$ and $R(S)$ corresponding to $S = cdefabcgabcde$. Here abc, cde, and c are the displayable entities of S. There are two outgoing re-edges from the vertex representing abc. These edges correspond to $x = d$ and $x = g$. $imp(S, abcd)$ $= imp(S, abcg) = S$. Consequently, both edges are incident on the sink. There are no edges corresponding to the other letters of the alphabet as $imp(S, abcx)$ does not exist for $x \in \{a, b, c, e, f\}$.

The space required for $CSD(S)$ is $O(n)$ and the time needed to construct it is $O(n)$ [7, 9]. While we have defined the csdawg data structure for a single string S, it can be extended to represent a set of strings.

```
Algorithm LinearOccurrences(S, v)
{ Find all occurrences of str(v) in S}
   Occurrences(S, v, 0)

Procedure Occurrences(S:string,u:vertex,i:integer)
1  begin
2     if str(u) is a suffix of S
3     then output(|S| − i);
4     for each re-edge e from u do
5        begin
6           Let w be the vertex on which e is incident;
7           Occurrences(S, w, |label(e)| + i);
8        end;
9  end;
```

Figure 3.2. Algorithm for obtaining all occurrences of displayable entity

3.1.2 Computing Occurrences of Displayable Entities in a string

Figure 3.2 presents an algorithm for computing the end positions of all the occurrences of $str(v)$ in S. This is based on the outline provided by Blumer et al [9]. First, the algorithm determines whether $str(v)$ is a suffix of S. If so, the end position of S is reported. The remaining occurrences of $str(v)$ are computed recursively by examining each node reached by an outgoing re-edge from v. The complexity of *LinearOccurrences*(S, v) is proportional to the number of occurrences of $str(v)$ in S.

3.1.3 Prefix and Suffix Extension Trees

The *prefix extension tree* $PET(S, v)$ at vertex v in $V(S)$ is a subgraph of $CSD(S)$ consisting of

(i) the root v

(ii) $PET(S, w)$ defined recursively for each vertex w in $V(S)$ such that there exists

a prefix extension edge from v to w, and

(iii) the prefix extension edges leaving v.

The *suffix extension tree* $SET(S, v)$ at v is defined analogously.

In Figure 3.1, $PET(S, v)$, where v is the vertex representing the substring c, consists of the vertices representing c and cde, and the sink. It also includes the prefix extension edges from c to cde and from cde to the sink. Similarly, $SET(S, v)$, where v is the vertex representing c, consists of the vertices representing c and abc and the suffix extension edge from c to abc (not shown in the figure).

Lemma 3.1.1 *$PET(S, v)$ ($SET(S, v)$) contains a directed path from v to a vertex w in $V(S)$ iff $str(v)$ is a prefix (suffix) of $str(w)$.*

Proof: If there is a directed path in $PET(S, v)$ from v to some vertex w, then from the definition of a prefix extension edge and the transitivity of the "prefix-of" relation, $str(v)$ must be a prefix of $str(w)$.

If $str(v)$ is a prefix of $str(w)$, then there exists a series of re-edges from v to w, such that $str(v)$, when concatenated with the labels on these edges yields $str(w)$. But, each of these re-edges must be a prefix extension edge. So a directed path from v to w exists in the $PET(S, v)$.

The proof for $SET(S, v)$ is analogous. ■

3.2 Computing Conflicts

3.2.1 Algorithm to determine whether a string is conflict-free

Before describing our algorithm to determine if a string is free of conflicts, we establish some properties of conflict-free strings that will be used in this algorithm.

Lemma 3.2.1 *If a prefix-suffix conflict occurs in a string S, then a subword conflict must occur in S.*

Proof: If a prefix-suffix conflict occurs between two displayable entities W_1 and W_2 in S, then there exists $W_pW_mW_s$ in S such that $W_pW_m = W_1$ and $W_mW_s = W_2$. Since W_1 and W_2 are maximal, W_1 isn't always followed by the same letter and W_2 isn't always preceded by the same letter; i.e., W_m isn't always followed by the same letter and W_m isn't always preceded by the same letter. So, W_m is maximal. But, W_1 occurs at least twice in S (since W_1 is a displayable entity). So W_m occurs at least twice (since W_m is a subword of W_1) and is a displayable entity of S. But, W_m is a subword of W_1. So a subword conflict occurs between W_m and W_1 in S. ■

Corollary 3.2.1 *The intersection of a prefix-suffix conflict between two displayable entiites is itself a displayable entity.*

Corollary 3.2.2 *If string S is free of subword conflicts, then it is free of conflicts.*

Lemma 3.2.2 *$str(w)$ is a subword of $str(v)$ in S iff there is a path comprising right extension and suffix extension edges from w to v.*

Proof: From the definition of $CSD(S)$, if there exists an re-edge from u to v, then $str(u)$ is a subword of $str(v)$. If there exists a suffix extension edge from u to v, then $str(u)$ is a suffix (and therefore a subword) of $str(v)$. If there exists a path comprising right and suffix extension edges from w to v, then by transitivity, $str(w)$ is a subword of $str(v)$.

If $str(w)$ is a suffix of $str(v)$, then there is a path (Lemma 3.1.1) of suffix extension edges from w to v. If $str(w)$ is a subword, but not a suffix of $str(v)$, then from

the definition of a csdawg, there is a path of re-edges from w to a vertex representing a suffix of $str(v)$. ■

Let V_{source} be the set of all vertices in $V(S)$ such that an re-edge or suffix extension edge exists between the source vertex of $CSD(S)$ and each element of V_{source}.

Lemma 3.2.3 *String S is conflict-free iff all right extension or suffix extension edges leaving vertices in V_{source} end at the sink vertex of $CSD(S)$.*

Proof: A string S is conflict-free iff there does not exist a right or suffix extension edge between two vertices, neither of which is the source or sink of $CSD(S)$ (Corollary 3.2.2 and Lemma 3.2.2).

Assume that S is conflict-free. Consider any vertex v in V_{source}. If v has right or suffix extension out edge $< v, w >$, then $v \neq sink$. If $w \neq sink$, then $str(v)$ is a subword of $str(w)$ and the string is not conflict-free. This contradicts the assumption on S.

Next, assume that all right and suffix extension edges leaving vertices in V_{source} end at the sink vertex. Clearly, there cannot exist right or suffix extension edges between any two vertices, v and w ($v \neq sink$, $w \neq sink$) in V_{source}. Further, there cannot exist a vertex x in $V(S)$ ($x \neq source$, $x \neq sink$) such that $x \notin V_{source}$. For such a vertex to exist, there must exist a path consisting of right and suffix extension edges from a vertex in V_{source} to x. Clearly, this is not true. So, S is conflict-free. ■

The preceding development leads to algorithm *NoConflicts* (Figure 3.3).

Theorem 3.2.1 *Algorithm NoConflicts is both correct and optimal.*

Algorithm *NoConflicts*(S)
1. Construct $CSD(S)$.
2. Compute V_{source}.
3. Scan all right and suffix extension out edges from each element of V_{source}. If any edge points to a vertex other than the sink, then a conflict exists. Otherwise, S is conflict-free.

Figure 3.3. Algorithm to determine whether a string is conflict-free

Proof: Correctness is an immediate consequence of Lemma 3.2.3. Step 1 takes $O(n)$ time [9]. Step 2 takes $O(1)$ time since $|V_{source}| < 2|\Sigma|$. Step 3 takes $O(1)$ time since the number of out edges leaving vertices in V_{source} is less than $4|\Sigma^2|$. So, *NoConflicts* takes $O(n)$ time, which is optimal. Actually, Steps 2 and 3 can be merged into Step 1 and the construction of $CSD(S)$ aborted as soon as an edge that violates Lemma 3.2.3 is created. ■

3.2.2 Subword Conflicts

Consider the problem of finding all subword conflicts in string S. Let k_s be the number of subword conflicts in S. Any algorithm to solve this problem requires (i) $O(n)$ time to read in the input string and (ii) $O(k_s)$ time to output all subword conflicts. So, $O(n + k_s)$ is a lower bound on the time complexity for this problem. For the string $S = a^n$, $k_s = n^4/24 + n^3/4 - 13n^2/24 - 3n/4 + 1 = O(n^4)$. This is an upper bound on the number of conflicts as the maximum number of substring occurrences is $O(n^2)$ and in the worst case, all occurrences conflict with each other. In this section, a compact method for representing conflicts is presented. Let k_{sc} be the size of this representation. k_{sc} is $n^3/6+n^2/2-5n/3$ or $O(n^3)$, for a^n. Compaction

never increases the size of the output and may yield up to a factor of n reduction, as in the example. The compaction method is described below.

Consider S= *abcdbcgabcdbchbc*. The displayable entities are D_1 = *abcdbc* and D_2 = *bc*. The end positions of D_1 are 6 and 13 while those of D_2 are 3, 6, 10, 13, and 16. A list of the subword conflicts between D_1 and D_2 can be written as {(6,3), (6,6), (13,10), (13,13)}. The first element of each ordered pair is the last position of the instance of the superstring (here D_1) involved in the conflict; the second element of each ordered pair is the last position of the instance of the substring (here D_2) involved in the conflict.

The cardinality of the set is the number of subword conflicts between D_1 and D_2. This is given by *frequency*(D_1)$*$*number of occurrences of* D_2 *in* D_1. Since each conflict is represented by an ordered pair, the size of the output is 2(*frequency*(D_1)$*$*number of occurrences of* D_2 *in* D_1).

Observe that the occurrences of D_2 in D_1 are in the same relative positions in all instances of D_1. It is therefore possible to write the list of subword conflicts between D_1 and D_2 as (6,13):(0,-3). The first list gives all the occurrences in S of the superstring (D_1), and the second gives the relative positions of all the occurrences of the substring (D_2) in the superstring (D_1) from the right end of D_1. The size of the output is now *frequency*(D_1)+*number of occurrences of* D_2 *in* D_1. This is more economical than our earlier representation.

In general, a substring D_i of S will have subword conflicts with many instances of a number of displayable entities (say, $D_j, D_k, \ldots, D_z$) of which it (D_i) is the superword. We would then write the conflicts of D_i as

$$(l_i^1, l_i^2, \ldots, l_i^{m_i}) : (l_j^1, l_j^2, \ldots, l_j^{m_j}), (l_k^1, l_k^2, \ldots, l_k^{m_k}), \ldots, (l_z^1, l_z^2, \ldots, l_z^{m_z}).$$

Here, the l_i's represent all the occurrences of D_i in S; the l_j's, l_k's, ..., l_z's represent the relative positions of all the occurrences of $D_j, D_k, \ldots, D_z$, respectively,

```
Algorithm SubwordConflicts(S:string)
{ Identify displayable entities that contain subword displayable entities}
1     begin
2         for each vertex v in CSD(S) do
3             begin
4                 v.subword = false;
5                 for all vertices u such that a right or suffix extension edge < u, v >
                  is incident on v do
6                     if u ≠ source then v.subword = true;
7             end
8         for each vertex v in CSD(S) such that v ≠ sink, v.subword is true do
9             GetSubwords(S, v);
10    end

Procedure GetSubwords(S, v)
{ Compute subword displayable entities of str(v)}
1     begin
2         LinearOccurrences(S,v);
3         v.sublist = {0};
4         SetUp(v);
5         SetSuffixes(v);
6         for each vertex, x (≠ source), in reverse topological order of SG(S, v) do
7         begin
8             if str(x) is a suffix of str(v) then x.sublist = {0} else x.sublist = {};
9             for each w in SG(S, v) on which an re-edge e from x is incident do
10                begin
11                    for each element l in w.sublist do
12                        x.sublist = x.sublist ∪ {l − |label(e)|};
13                end;
14            output(x.sublist);
15        end;
16    end
```

Figure 3.4. Optimal algorithm to compute all subword conflicts

in D_i. One such list will be required for each displayable entity that contains other displayable entities as subwords. The following equalities are easily obtained:

Size of Compact Representation $= \sum_{D_i \epsilon D} (f_i + \sum_{D_j \epsilon D_i^s} (r_{ij}))$.

Size of Original Representation $= 2 \sum_{D_i \epsilon D} (f_i * \sum_{D_j \epsilon D_i^s} (r_{ij}))$.

Here, f_i is the frequency of D_i (only D_i's that have conflicts are considered), and r_{ij} is the frequency of D_j in one instance of D_i. The letter D represents the set of all displayable entities of S while D_i^s represents the set of all displayable entities that are subwords of D_i.

$SG(S,v)$, for $v \epsilon V(S)$, is defined as the subgraph of $CSD(S)$ which consists of the set of vertices $SV(S,v) \subset V(S)$ which represents displayable entities that are subwords of $str(v)$ and the set $SE(S,v)$ of all re-edges and suffix extension edges that connect any pair of vertices in $SV(S,v)$. Define $SG_R(S,v)$ as $SG(S,v)$ with the directions of all the edges in $SE(S,v)$ reversed.

Lemma 3.2.4 $SG(S,v)$ *consists of all vertices* w *such that a path comprising right or suffix extension edges joins* w *to* v *in* $CSD(S)$.

Proof: Follows from Lemma 3.2.2 and the definition of $SG(S,v)$. ■

Algorithm *SubwordConflicts(S)* of Figure 3.4 computes all subword conflicts of S. The subword conflicts are computed for precisely those displayable entities which have subword displayable entities. Lines 4 to 6 of *SubwordConflicts* determine whether $str(v)$ has subword displayable entities. Each incoming right or suffix extension edge to v is checked to see whether it originates at the source. If any incoming edge originates at a vertex other than *source*, then *v.subword* is set to **true** (Lemma 3.2.2). If all incoming edges originate from *source*, then *v.subword* is set

to **false**. **Procedure** *GetSubwords*(S, v), which computes the subword conflicts of $str(v)$ is invoked iff $v.subword$ is **true**.

Procedure *LinearOccurrences*(S, v) (line 2 of *GetSubwords*) reports the occurrences of $str(v)$ in S. **Procedure** *SetUp(v)* in line 4 traverses $SG_R(S, v)$ and initializes fields in each vertex of $SG_R(S, v)$ so that a reverse topological traversal of $SG(S, v)$ may subsequently be performed. **Procedure** *SetSuffixes(v)* in line 5 marks vertices whose displayable entities are suffixes of $str(v)$. This is accomplished by following the chain of reverse suffix extension pointers starting at v and marking the vertices encountered as suffixes of v.

A list of relative occurrences *sublist* is associated with each vertex x in $SG(S, v)$. $x.sublist$ represents the relative positions of $str(x)$ in an occurrence of $str(v)$. Each relative occurrence is denoted by its position relative to the last position of $str(v)$ which is represented by 0. If $str(x)$ is a suffix of $str(v)$ then $x.sublist$ is initialized with the element 0. The remaining elements of $x.sublist$ are computed from the *sublist* fields of vertices w in $SG(S, v)$ such that a right extension edge goes from x to w. Consequently, $w.sublist$ must be computed before $x.sublist$. This is achieved by traversing $SG(S, v)$ in reverse topological order [10].

Lemma 3.2.5 *$x.sublist$ for vertex x in $SG(S, v)$ contains all relative occurrences of $str(x)$ in $str(v)$ on completion of GetSubwords(S,v).*

Proof: The correctness of this lemma follows from the correctness of procedure *LinearOccurrences*(S, v) of Section 3.1.2 and the observation that lines 6 to 14 of procedure *GetSubwords* achieve the same effect as *LinearOccurrences*(S, v) in $SG(S, v)$. ■

Theorem 3.2.2 Algorithm *SubwordConflicts takes $O(n + k_{sc})$ time and space and is therefore optimal.*

Proof: Computing *v.subword* for each vertex v in $V(S)$ takes $O(n)$ time as constant time is spent at each vertex and edge in $CSD(S)$. Consider the complexity of *GetSubwords*(S, v). Line 2 takes $O(|v.occurrences|)$ time. Let the number of vertices in $SG(S, v)$ be m. Then the number of edges in $SG(S, v)$ is $O(m)$ since each vertex has at most $2|\Sigma| = O(1)$ edges leaving it. Line 4 traverses $SG(S, v)$ and therefore consumes $O(m)$ time. Line 5, in the worst case, could involve traversing $SG(S, v)$ which takes $O(m)$ time. Computing the relative occurrences of $str(x)$ in $str(v)$ (lines 8-14) takes $O(|x.sublist|)$ time for each vertex x in $SG(S, v)$. So, the total complexity of *GetSubwords*(S, v) is $O(|v.occurrences| + m + \Sigma_{x \epsilon SV(S,v)-v}|x.sublist|)$.

However, m is $O(\sum_{x \epsilon SV(S,v)-\{v\}} |x.sublist|)$, since $|x.sublist| \geq 1$ for each $x \epsilon SG(S, v)$. But $|v.occurrences| + \sum_{x \epsilon SV(S,v)-\{v\}} |x.sublist|$ is the size of the output of *GetSubwords*(S, v).

So, the complexity of Algorithm *SubwordConflicts(S)* is $O(n+\Sigma_{v \epsilon V(S)-\{sink\}, v.subword=true}|$ output of *GetSubwords*$(S, v)|) = O(n + k_{sc})$. ■

3.2.3 Prefix-Suffix Conflicts

As with subword conflicts, the lower bound for the problem of computing prefix-suffix conflicts is $O(n + k_p)$, where k_p is the number of prefix-suffix conflicts in S. For $S = a^n$, k_p is $n^4/24 - n^3/12 - 25n^2/24 - 21n/12 + 1 = O(n^4)$, which is also the upper bound on k_p. Unlike subword conflicts, it is not possible to compact the output representation.

Let w and x, respectively, be vertices in $SET(S, v)$ and $PET(S, v)$. Let $str(v) = W_v$, $str(w) = W_w W_v$, and $str(x) = W_v W_x$. Define $Pshadow(w, v, x)$ to be the vertex

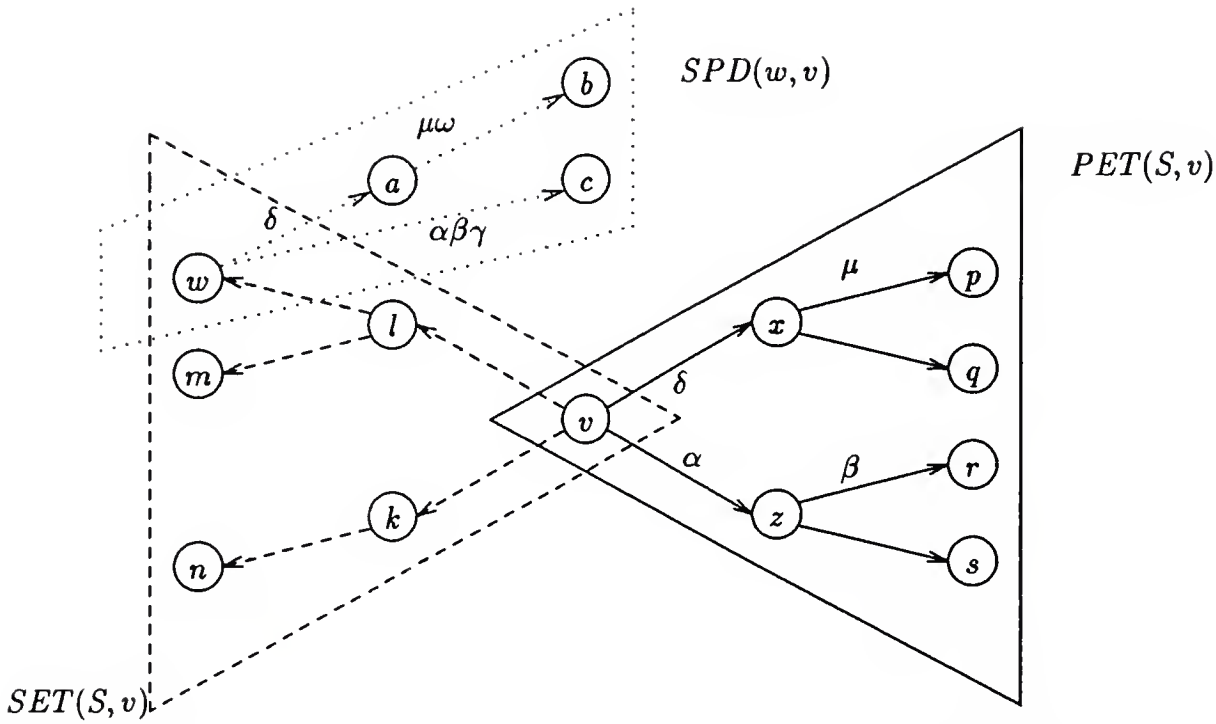


Figure 3.5. Illustration of prefix and suffix trees and a shadow prefix dag

representing $imp(S, W_wW_vW_x)$, if such a vertex exists. Otherwise, $Pshadow(w, v, x)$ = nil. We define *Pimage(w,v,x) = Pshadow(w,v,x)* iff *Pshadow(w,v,x) = imp(S,*$W_wW_vW_x$*)* = $W_aW_wW_vW_x$ for some (possibly empty) string W_a. Otherwise, $Pimage(w, v, x)$ = nil. For each vertex w in $SET(S, v)$, a *shadow prefix dag* $SPD(w, v)$ rooted at vertex w comprises the set of vertices $\{Pshadow(w, v, x) |\ x$ on $PET(S, v)$, $Pshadow(w, v, x)$ $\neq$ nil$\}$.

Figure 3.5 illustrates these concepts. Broken lines represent suffix extension edges, dotted lines represent right extension edges, and solid lines represent prefix extension edges. $SET(S, v)$, $PET(S, v)$, and $SPD(w, v)$ have been enclosed by dashed, solid, and dotted lines, respectively. We have $Pshadow(w, v, v) = Pimage(w, v, v)$

$= w$. $Pshadow(w, v, z) = Pshadow(w, v, r) = c$. However, $Pimage(w, v, z) = Pimage(w, v, r) = nil$. $Pshadow(w, v, x) = Pimage(w, v, x) = a$. $Pshadow(w, v, p) = b$, but $Pimage(w, v, p) = nil$. $Pshadow(w, v, q) = Pshadow(w, v, s) = Pimage(w, v, q) = Pimage(w, v, s) = nil$.

Lemma 3.2.6 *A prefix-suffix conflict occurs between two displayable entities, $W_1 = str(w)$ and $W_2 = str(x)$, with respect to a third displayable entity $W_m = str(v)$ iff (i) w occurs in $SET(S, v)$ and x occurs in $PET(S, v)$, and (ii) $Pshadow(w, v, x) \neq nil$. The number of conflicts between $str(w)$ and $str(x)$ with respect to $str(v)$ is equal to the number of occurrences of $str(Pshadow(w, v, x))$ in S.*

Proof: By definition, a prefix-suffix conflict occurs between displayable entities W_1 and W_2 with respect to W_m iff there exists $W_pW_mW_s$ in S, where $W_1 = W_pW_m$ and $W_2 = W_mW_s$.

Clearly, W_m is a suffix of W_1 and W_m a prefix of W_2 iff w occurs in $SET(S, v)$ and x occurs in $PET(S, v)$. $W_pW_mW_s$ occurs in S iff $imp(S, W_pW_mW_s)$ exists or $Pshadow(w, v, x) \neq nil$. The number of conflicts between $str(w)$ and $str(x)$ is equal to the number of occurrences of $imp(S, W_pW_mW_s) = str(Pshadow(w, v, x))$ in S. ■

Lemma 3.2.7 *If a prefix-suffix conflict does not occur between $str(w)$ and $str(x)$ with respect to $str(v)$, where w occurs in $SET(S, v)$ and x occurs in $PET(S, v)$, then there are no prefix-suffix conflicts between any displayable entity which represents a descendant of w in $SET(S, v)$ and any displayable entity which represents a descendant of x in $PET(S, v)$ with respect to $str(v)$.*

Proof: Since w is in $SET(S, v)$ and x is in $PET(S, v)$, we can represent $str(w)$ by $W_pstr(v)$ and $str(x)$ by $str(v)W_s$. If no conflicts occur, then $W_pstr(v)W_s$ does

not occur in S. The descendants of w in $SET(S,v)$ will represent displayable entities of the form $W_a str(w) = W_a W_p str(v)$, while the descendants of x in $PET(S,v)$ will represent displayable entities of the form $str(x)W_b = str(v)W_s W_b$, where W_a and W_b are substrings of S. For a prefix-suffix conflict to occur between $W_a str(w)$ and $str(x)W_b$ with respect to $str(v)$, $W_a W_p str(v) W_s W_b$ must exist in S. However, this is not possible as $W_p str(v) W_s$ does not occur in S and the result follows. ■

Lemma 3.2.8 *In $CSD(S)$, if*

(i) $y = Pimage(w,v,x)$,

(ii) there is a prefix extension edge e from x to z with label $a\alpha$, and

(iii) there is a right extension edge f from y to u with label $a\beta$,

then $Pshadow(w,v,z) = u$.

Proof: Let $str(w) = W_w str(v)$ and $str(x) = str(v)W_x$. By definition, $str(y) = W_a W_w str(v) W_x$ for some, possibly empty, string W_a. We have $str(z) = str(x)a\alpha = str(v)W_x a\alpha$ and $str(u) = W_b str(y) a\beta = W_b W_a W_w str(v) W_x a\beta$ for some string W_b.

$Pshadow(w,v,z) = imp(W_w str(v) W_x a\alpha)$. To prove the lemma, we must show that $Pshadow(w,v,z) = u$. Or, that (i) $W_w str(v) W_x a\alpha$ is a subword of $str(u)$ and (ii) $str(u)$ is the smallest superword of $W_w str(v) W_x a\alpha$ represented by a vertex in $CSD(S)$.

(*i*) Assume that $W_w str(v) W_x a\alpha$ is not a subword of $str(u) = W_b W_a W_w str(v) W_x a\beta$. So, α is not a prefix of β.

Case 1: β is a proper prefix of α.

Since $W_b W_a W_w str(v) W_x a\beta$ is maximal, its occurrences are not all followed by the same letter. This statement is also true for any of its suffixes. In particular, all

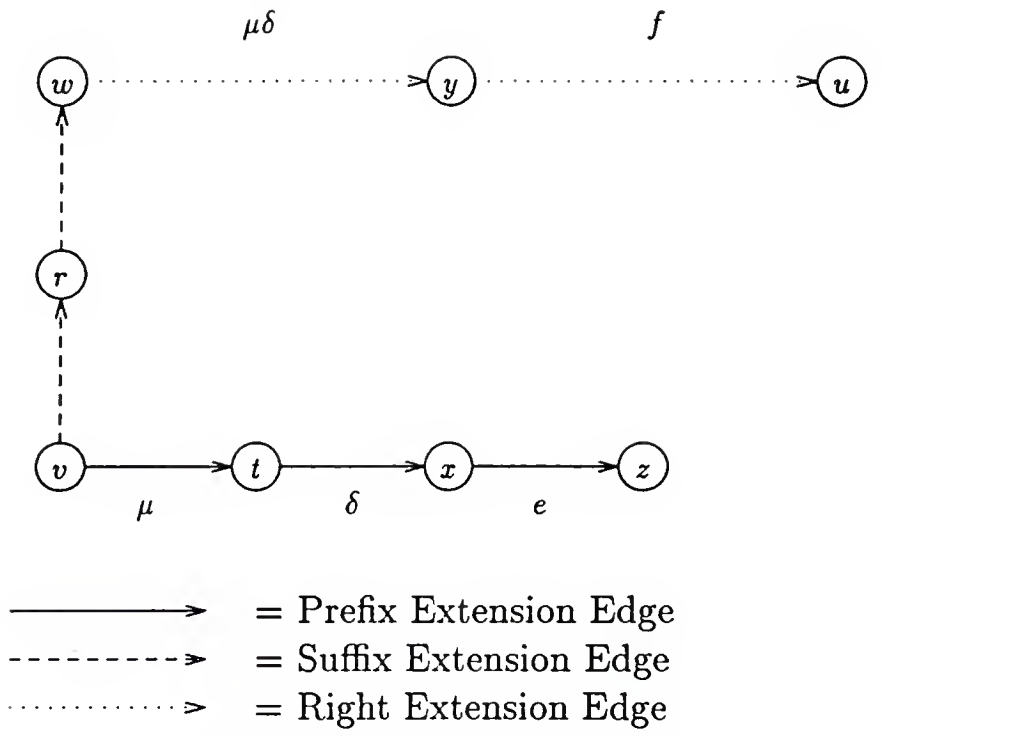


Figure 3.6. Illustration of conditions for Lemma 3.2.8

occurrences of $str(v)W_xa\beta$ cannot be followed by the same letter. Similarly, all occurrences of $str(v)W_xa\beta$ cannot be preceded by the same letter as it is a prefix of $str(v)W_xa\alpha = str(z)$. So, $str(v)W_xa\beta$ is a displayable entity of S. Consequently, the prefix extension edge from x corresponding to the letter a must be directed to the vertex representing $str(v)W_xa\beta$. This is a contradiction.

Case 2: $a\beta$ matches $a\alpha$ in the first k characters, but not in the $(k+1)'th$ character $(1 \leq k < 1+min(|\alpha|,|\beta|))$.

We have $a\beta = a\gamma\beta_1$, $a\alpha = a\gamma\alpha_1$, where $|\gamma| = k-1$. Clearly, the strings $str(v)W_xa\gamma\alpha_1$ and $W_bW_aW_wstr(v)W_xa\gamma\beta_1$ occur in S. So, all occurrences of $str(v)W_xa\gamma$ cannot be followed by the same letter. Further, all occurrences of $str(v)W_xa\gamma$ cannot be preceded by the same letter as it is a prefix of $str(v)W_xa\alpha = str(z)$. So, it is a displayable entity of S. Consequently, the prefix extension edge from x corresponding to the letter a must be directed to the vertex representing $str(v)W_xa\gamma$. This results in a contradiction. Thus, α is a prefix of β.

(ii) From (i), α is a prefix of β. Assume that $W_bW_aW_wstr(v)W_xa\beta$ is not the smallest superword of $W_wstr(v)W_xa\alpha$. Since $str(y) = Pimage(w,v,x) = W_aW_wstr(v)W_x$ is the smallest superword of $W_wstr(v)W_x$, the smallest superword of $W_wstr(v)W_xa\alpha$ must be of the form $W_{b_1}W_aW_wstr(v)W_xa\gamma$ where α is a prefix of γ which is a proper prefix of β and/or W_{b_1} is a proper suffix of W_b. But, the right out edge f from z points to the smallest superword of $W_aW_wstr(v)W_xa$ (from the definition of $CSD(S)$) which is $W_bW_aW_wstr(v)W_xa\beta$. So, $W_{b_1} = W_b$ and $\gamma = \beta$, which is a contradiction. ■

Lemma 3.2.9 *In $CSD(S)$, if*

(i) $y = Pimage(w,v,x)$,

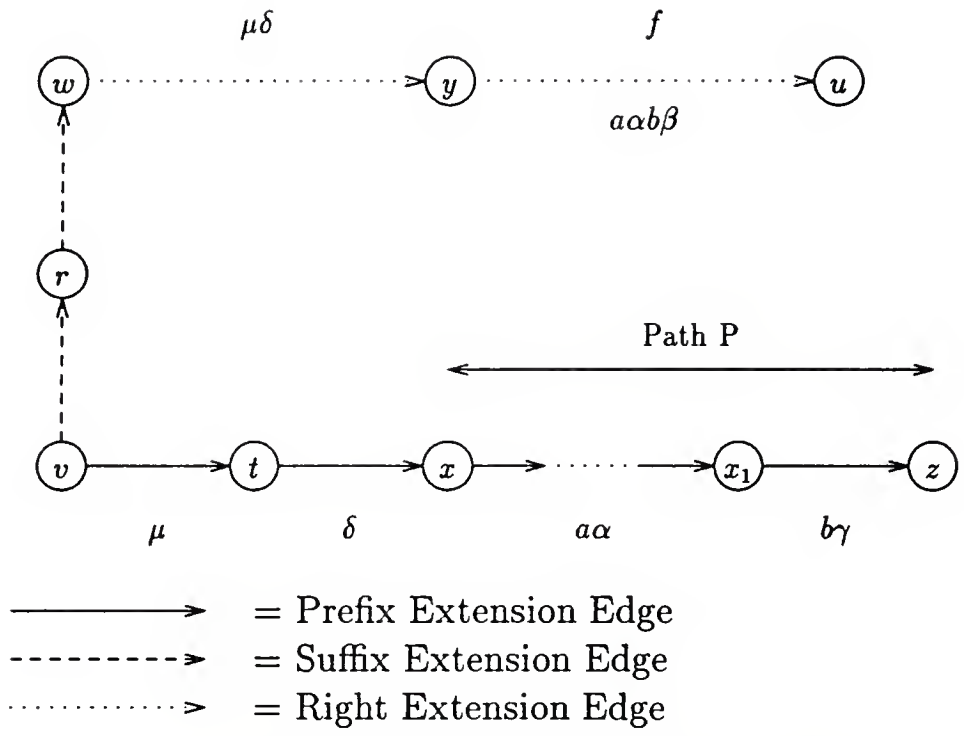


Figure 3.7. Illustration of conditions for Lemmas 3.2.9 and 3.2.10

(ii) there is a path of prefix extension edges from x to x_1 (let the concatenation of their labels be $a\alpha$),
(iii) there is a prefix extension edge from x_1 to z with label $b\gamma$, and
(iv) there is a right extension edge f from y to u with label $a\alpha b\beta$,
then $u = Pshadow(w, v, z) \neq nil$.

Proof: Similar to proof of Lemma 3.2.8. ■

Lemma 3.2.10 *In Lemma 3.2.8 or Lemma 3.2.9, if $|label(f)| \leq$ sum of the lengths of the labels of of the edges on the prefix extension edge path P from x to z, then $label(f)$ = concatenation of the labels on P and $u = Pimage(w, v, z)$.*

Proof: From Lemma 3.2.9, the concatenation of the labels of the edges of P is a prefix of $label(f)$. But, $|label(f)| \leq$ sum of the lengths of the labels of the edges on P. Thus, $label(f)$ = concatenation of the labels of the series edges on P. $str(u) = Pimage(w, v, z)$ follows. ■

Lemma 3.2.11 *If $Pshadow(w, v, x) = nil$ then $Pshadow(w, v, y) = nil$ for all descendants y of x in $PET(S, v)$.*

Proof: Follows from Lemmas 3.2.6 and 3.2.7. ■

Algorithm *PrefixSuffixConflicts(S)* in Figure 3.8 computes all prefix-suffix conflicts in S. Line 1 constructs $CSD(S)$. Lines 2 and 3 compute all prefix-suffix conflicts in S by separately computing for each displayable entity $str(v)$, all the prefix-suffix conflicts of which it is the intersection (Corollary 3.2.1).

Algorithm *PrefixSuffixConflicts(S:string)*
{Compute all prefix-suffix conflicts in S}
1 Construct $CSD(S)$.
2 **for** each vertex v in $CSD(S)$ **do**
3 *NextSuffix(v,v)*; {compute all conflicts wrt $str(v)$}

Procedure *NextSuffix(current,v: vertex)*;
1 **for** each suffix extension edge $< current, w >$ **do**
2 {there can only be one suffix extension edge from *current* to w}
3 **begin**
4 *exist* = false
5 *ShadowSearch*(v, w, v, w);{compute $SPD(w, v)$}
6 **if** *exist* **then** *NextSuffix(w,v)*;
7 **end**;

Figure 3.8. Optimal algorithm to compute all prefix-suffix conflicts

Procedure *NextSuffix(current,v)* computes all prefix-suffix conflicts between displayable entities represented by descendants of *current* in $SET(S, v)$ and displayable entities represented by descendants of v in $PET(S, v)$ with respect to $str(v)$ (so the call to *NextSuffix(v,v)* in line 3 of **Algorithm** *PrefixSuffixConflicts(S)* computes all prefix-suffix conflicts with respect to $str(v)$). It does so by identifying $SPD(w, v)$ for each child w of *current* in $SET(S, v)$. The call to *ShadowSearch(v,w,v,w)* in line 5 identifies $SPD(w, v)$ and computes all prefix-suffix conflicts between $str(w)$ and displayable entities represented by descendants of v in $PET(S, v)$ with respect to $str(v)$. If *ShadowSearch(v,w,v,w)* does not report any prefix-suffix conflicts then the global variable *exist* is unchanged by *ShadowSearch(v,w,v,w)* (so, *exist* = **false**, from line 4). Otherwise, it is set to **true** by *ShadowSearch*. Line 6 ensures that *NextSuffix(w,v)* is called only if *ShadowSearch(v,w,v,w)* detected prefix-suffix conflicts between $str(w)$ and displayable entities represented by descendants of v in $PET(S, v)$ with respect to $str(v)$ (Lemma 3.2.7).

For each descendant q of vertex x in $PET(S, v)$, procedure *ShadowSearch(v,w,x,y)* computes all prefix-suffix conflicts between $str(w)$ and $str(q)$ with respect to $str(v)$. y represents $Pshadow(w, v, x)$. We will show that all calls to *ShadowSearch* maintain the invariant (which is referred to as the *image invariant* hereafter) that $y = Pimage(w, v, x) \neq nil$. Notice that the invariant holds when *ShadowSearch* is called from *NextSuffix* as $w = Pimage(w, v, v)$. The **for** statement in line 1 examines each prefix extension edge leaving x. Lines 3 to 28 compute all prefix-suffix conflicts between $str(w)$ and displayable entities represented by vertices in $PET(S, z)$, where z is the vertex on which the prefix extension edge from x is incident. The truth of the condition in the **for** statement of line 1, line 4, and the truth of the condition inside the **if** statement of line 5 establish that the conditions of Lemma 3.2.8 are satisfied prior to the execution of lines 8 and 9. The truth of the comment in line 8

```
Procedure ShadowSearch(v, w, x, y);
1   for each prefix extension edge e = < x, z > do
2   {There can only be one prefix extension edge from x to z}
3       begin
4           fc:= first character in label(e);
5           if there is a right extension edge, f = < y, u >, whose label starts with fc
6           then
7               begin
8                   {u = Pshadow(w, v, x)}
9                   ListConflicts(u,z,w);
10                  distance:= 0; done = false
11                  while (not done) and (|label(f)| > |label(e)| + distance)) do
12                      begin
13                          distance:= distance + |label(e)|;
14                          nc:= (distance + 1)^th character in label(f).
15                          if there is a prefix extension edge < z, r > starting with nc
16                          then
17                              begin
18                                  z := r;
19                                  {u = Pshadow(w, v, z)};
20                                  ListConflicts(u,z,w);
21                              end
22                          else
23                              done:= true;
24                      end
25                  if (not done) then
26                      ShadowSearch(v,w,z,u);
27                  exist:= true;
28              end
29      end
```

Figure 3.9. Algorithm for shadow search

and the correctness of line 9 are established by Lemma 3.2.8. **Procedure** *ListConflicts* of line 9 lists all prefix-suffix conflicts between $str(w)$ and $str(z)$ with respect to $str(v)$. Similarly, the truth of the condition inside the **while** statement of line 11, lines 13 and 14, and the truth of the condition inside the **if** statement of line 15 establish that the conditions of Lemma 3.2.9 are satisfied prior to the execution of lines 18-20. Again, the correctness of lines 18-20 are established by Lemma 3.2.9. If *done* remains **false** on exiting the while loop, the condition of the **if** statement of line 15 must have evaluated to **true**. Consequently, the conditions of Lemma 3.2.9 apply. Further, since the **while** loop of line 11 terminated, the additional condition of Lemma 3.2.10 is also satisfied. Hence, from Lemma 3.2.10, $u = Pimage(w, v, z)$ and the image invariant for the recursive call to $ShadowSearch(v, w, z, u)$is maintained. Line 27 sets the global variable *exist* to **true** since the execution of the **then** clause of the **if** statement of line 5 ensures that at least one prefix-suffix conflict is reported by $ShadowSearch(v, w, v, w)$ (Lemmas 3.2.6 and 3.2.8). *exist* remains **false** only if the **then** clause of the **if** statement (line 5) is never executed.

Theorem 3.2.3 **Algorithm** *PrefixSuffixConflicts(S) computes all prefix-suffix conflicts of* S *in* $O(n + k_p)$ *space and time, which is optimal.*

Proof: Line 1 of **Algorithm** *PrefixSuffixConflicts* takes $O(n)$ time [9]. The cost of lines 2 and 3 without including the execution time of $NextSuffix(v, v)$ is $O(n)$.

Next, we show that $NextSuffix(v, v)$ takes $O(k_v)$ time, where k_v is the number of prefix-suffix conflicts with respect to v (so, k_v represents the size of the output of $NextSuffix(v, v)$). Assume that *NextSuffix* is invoked p times in the computation. Let S_T be the set of invocations of *NextSuffix* which do not call *NextSuffix* recursively. Let $p_T = |S_T|$. Let S_F be the set of invocations of *NextSuffix* which do call *NextSuffix* recursively. Let $p_F = |S_F|$. Each element of S_F can directly call at most $|\Sigma|$ elements

of S_T. So, $p_T/p_F \leq |\Sigma|$. From lines 4-6 in *NextSuffix(current,v)*, each element of S_F yields at least one distinct conflict from its call to *ShadowSearch*. Thus, $p_F \leq k_v$. So, $p = p_T + p_F \leq (|\Sigma| + 1)k_v = O(k_v)$. The cost of execution of *NextSuffix* without including the costs of recursive calls to *NextSuffix* and *ShadowSearch* is $O(|\Sigma|)$ $(= O(1))$ as there are at most $|\Sigma|$ suffix edges leaving a vertex. So, the total cost of execution of all invocations of *NextSuffix* spawned by *NextSuffix*(v, v) without including the cost of recursive calls to *ShadowSearch* is $O(p|\Sigma|) = O(k_v)$.

Next, we consider the calls to *ShadowSearch* that were spawned by *NextSuffix*(v, v). Let T_A be the set of invocations of *ShadowSearch* which do not call *ShadowSearch* recursively. Let $q_A = |T_A|$. Let T_B be the set of invocations of *ShadowSearch* which do call *ShadowSearch* recursively. Let $q_B = |T_B|$. Let $q = q_A + q_B$. We have $q_A \leq |\Sigma|q_B + |\Sigma|p$. So, $q = q_A + q_B \leq (|\Sigma|+1)q_B+|\Sigma|p$. From the algorithm, each element of T_B yields a distinct conflict. So, $q_B \leq k_v$. So, $q \leq (|\Sigma| + 1)q_B + |\Sigma|p = O(k_v)$. The cost of execution of a single call to *ShadowSearch* without including the cost of executing recursive calls to *ShadowSearch* is $O(1) + O(w) + O($*complexity of ListConflicts of line 9*$) + O(\sum_{i=1}^{w}($*complexity of ListConflicts of line 20 in the* i^{th} *iteration of the* **while** *loop*$))$, where w denotes the number of iterations of the **while** loop. The complexity of *ListConflicts* is proportional to the number of conflicts it reports. Since *ListConflicts* always yields at least one distinct conflict, the complexity of *ShadowSearch* is $O(1+|output|)$. Summing over all calls to *ShadowSearch* spawned by *NextSuffix*(v, v), we obtain $O(q + k_v) = O(k_v)$. Thus, the total complexity of **Algorithm** *PrefixSuffixConflicts(S)* is $O(n + k_p)$ ■

3.2.4 Alternative Algorithms

In this section, an algorithm for computing *all* conflicts (i.e., both subword and prefix-suffix conflicts) is presented. This solution is relatively simple and has competitive run times. However, it lacks the flexibility required to efficiently solve many of the problems listed in Sections 3.3, 3.4, and 3.5 . The algorithm (**Algorithm** *AllConflicts(S)*) is presented in Figure 3.10. Step 1 computes a list of all occurrences of all displayable entities in S. This list is obtained by first computing the lists of occurrences corresponding to each vertex of $V(S)$ (except the source and the sink) and then concatenating these lists. Each occurrence is represented by its start and end positions. Step 2 sorts the list of occurrences obtained in Step 1 in increasing order of their start positions. Occurrences with the same start positions are sorted in decreasing order of their end positions. This is done using radix sort. Step 3 computes for the i'th occurrence occ_i all its prefix-suffix conflicts with occurrences whose start positions are greater than its own, and all its subword conflicts with its subwords. occ_i is checked against occ_{i+1}, occ_{i+2},..., occ_{i+c} for a conflict. Here, c is the smallest integer for which there is no conflict between occ_i and occ_{i+c}. The start position of occ_{i+c} is greater than the end position of occ_i. The start position of occ_j ($j \geq i + c$) will also be greater than the end position of occ_i, since the list of occurrences was sorted on increasing order of start positions. The start positions of occ_{i+1},.., occ_{i+c-1} are greater than or equal to the start positions of occ_i but are less than or equal to its end position. Those occurrences among $\{occ_{i+1}, ..., occ_{i+c-1}\}$ whose start positions are equal to that of occ_i have end positions that are smaller (since occurrences with the same start position are sorted in decreasing order of their end positions). The remaining conflicts of occ_i (i.e., subword conflicts with its superwords, prefix-suffix conflicts with occurrences whose start positions are less than

that of occ_i) have already been computed in earlier iterations of the **for** statement in **Algorithm** *AllConflicts(S)*.

For example, let the input to Step 3 be the following list of ordered pairs:((1,6), (1,3), (1,1), (2,2), (3,8), (3,5), (4,6), (5,8), (6,10)),where the first element of the ordered pair denotes the start position and the second element denotes the end position of the occurrence. Consider the occurrence (3,5). Its conflicts with (1,6), (1,3), and (3,8) are computed in iterations 1, 2, and 5 of the **for** loop. Its conflicts with (4,6) and (5,8) are computed in iteration 6 of the for loop.

Theorem 3.2.4 **Algorithm** *AllConflicts(S) takes* $O(n+k)$ *time, where* $k = k_p + k_s$.

Proof: Step 1 takes $O(n+o)$ time, where o is the number of occurrences of displayable entities of S. Step 2 also takes $O(n+o)$ time, since o elements are to be sorted using radix sort with n buckets. Step 3 takes $O(o+k)$ time (the **for** loop executes $O(o)$ times; each iteration of the while loop yields a distinct conflict). So, the total complexity is $O(n+o+k)$.

We now show that $o = O(n+k)$. Let o_1 be the number of occurrences not involved in a conflict. Then $o_1 \leq n$. Let o_2 be the number of occurrences involved in at least one conflict. A single conflict occurs between two occurrences. So $2k \geq o_2$. So, $o = o_1 + o_2 \leq n + 2k = O(n+k)$. ■

Algorithm *AllConflicts(S)* can be modified so that the size of the output is $k_p + k_{sc}$. This may be achieved by checking whether an occurrence is the first representative of its pattern in the **for** loop of Step 3. The subword conflicts are only reported for the first occurrence of the pattern. However, the time complexity of **Algorithm** *AllConflicts(S)* remains $O(n+k)$. In this sense, it is suboptimal.

Algorithm *AllConflicts(S)*
Step 1: Obtain a list of *all* occurrences of *all* displayable entities in S. This list is obtained by first computing the lists of occurrences corresponding to each vertex of the csdawg (except the source and the sink) and then concatenating these lists.
Step 2: Sort the list of occurrences using the start positions of the occurrences as the primary key (increasing order) and the end position as the secondary key (decreasing order). This is done using radix sort.
Step3:

```
for i:= 1 to (number of occurrences) do
   begin
      j:= i + 1;
      while(lastpos(occ_i) ≥ firstpos(occ_j) do
         begin
            if (lastpos(occ_i) ≥ lastpos(occ_j))
            then occ_i is a superword of occ_j
            else (occ_i, occ_j) have a prefix-suffix conflict;
            j:= j + 1;
         end;
   end;
```

Figure 3.10. A simple algorithm for computing conflicts

3.3 Size Restricted Queries

Experimental data show that random strings contain a large number of displayable entities of small length. In most applications, small displayable entities are less interesting than large ones. Hence, it is useful to list only those displayable entities whose lengths are greater than some integer k. Similarly, it is useful to report exactly those conflicts in which the conflicting displayable entities have length greater than k. This gives rise to the following problems:

P8: List all occurrences of displayable entities whose lengths are greater than k.

P9: Compute all prefix-suffix conflicts involving displayable entities of length greater than k.

P10: Compute all subword conflicts involving displayable entities of length greater than k.

The *overlap* of a conflict is defined as the string common to the conflicting displayable entities. The overlap of a subword conflict is the subword displayable entity. The overlap of a prefix-suffix conflict is its intersection. The *size* of a conflict is the length of the overlap. An alternative formulation of the size restricted problem which also seeks to achieve the goal outlined above is based on reporting only those conflicts whose *size* is greater than k. This formulation of the problem is particularly relevant when the conflicts are of more interest than the displayable entities. It also establishes that all conflicting displayable entities reported have size greater than k. We have the following problems:

P11: Obtain all prefix-suffix conflicts of size greater than some integer k.

P12: Obtain all subword conflicts of size greater than some integer k.

P8 is solved optimally by invoking $LinearOccurrences(S, v)$ for each vertex v in $V(S)$, where $|str(v)| > k$. A combined solution to **P9** and **P10** uses the approach

of Section 3.2.4. The only modification to the algorithm of Figure 3.10 is in Step 1 which now becomes:

Obtain all occurrences of displayable entities whose lengths are greater than k.

The resulting algorithm is optimal with respect to the expanded representation of subword conflicts. However, as with the general problem, it is not possible to obtain separate optimal solutions to **P9** and **P10** by using the techniques of Section 3.2.4. An optimal solution to **P11** is obtained by executing line 3 of **Algorithm** *PrefixSuffixConflicts(S)* of Figure 3.8 for only those vertices v in $V(S)$ which have $|str(v)| > k$. An optimal solution to **P12** is obtained by the following modifications to **Algorithm** *SubwordConflicts* of Figure 3.4:

(*i*) Right extension or suffix extension edges $< u, v >$, where $|str(u)| \leq k$ and $|str(v)| > k$ are marked "disabled."

(*ii*) The definition of $SG(S, v)$ is modified so that $SG(S, v)$, for $v \epsilon V(S)$, is defined as the subgraph of $CSD(S)$ which consists of the set of vertices $SV(S, v) \subset V(S)$ which represent displayable entities of length greater than k that are subwords of $str(v)$ and the set of all re and suffix extension edges that connect any pair of vertices in $SV(S, v)$.

(*iii*) **Algorithm** *SubwordConflicts(S)* is modified. The modified algorithm is shown in Figure 3.11.

We note that **P9** and **P12** are identical, since the overlap of a subword conflict is the same as the subword displayable entity.

3.4 Pattern Oriented Queries

These queries are useful in applications where the fact that two patterns have a conflict is more important than the number and location of conflicts. The following problems arise as a result:

```
Algorithm SubwordConflicts(S,k)
1     begin
2         for each vertex v in CSD(S) do
3             v.subword = false;
4         for each vertex v in CSD(S) such that |str(v)| > k do
5             for all vertices u such that a non-disabled right or suffix extension edge
              < u, v > exists do
6                 v↑.subword = true;
7         for each vertex v in CSD(S) such that v ≠ sink and v.subword is true do
8             GetSubwords(S, v);
9     end
```

Figure 3.11. Modified version of algorithm SubwordConflicts

P13: List all pairs of displayable entities which have subword conflicts.

P14: List all triplets of displayable entities (D_1,D_2,D_m) such that there is a prefix-suffix conflict between D_1 and D_2 with respect to D_m.

P15: Same as **P13**, but size restricted as in **P12**.

P16: Same as **P14**, but size restricted as in **P11**.

P13 may be solved optimally by reporting for each vertex v in $V(S)$, where v does not represent the sink of $CSD(S)$, the subword displayable entities of $str(v)$, if any. This is accomplished by reporting $str(w)$, for each vertex w, $w \neq source$, in $SG(S, v)$. **P14** may also be solved optimally by modifying procedure *ListConflicts* of Figure 3.9 so that it reports the conflicting displayable entiities and their intersection. **P15** and **P16** may also be solved by making similar modifications to the algorithms of the previous section.

3.5 Statistical Queries

These queries are useful when conclusions are to be drawn from the data based on statistical facts. Let $f(D)$ denote the frequency (number of occurrences) of displayable entity D in the string and $rf(D_1, D_2)$ the number of occurrences of displayable entity D_1 in displayable entity D_2. The following queries may then be defined.

P17: For each pair of displayable entities, D_1 and D_2, involved in a subword conflict (D_1 is the subword of D_2), obtain $p(D_1, D_2)$ = (*number of occurrences of* D_1 *which occur as subwords of* D_2) / $f(D_1)$.

P18: For each pair of displayable entities, D_1 and D_2, involved in a prefix-suffix conflict, obtain $q(D_1, D_2)$ = (*number of occurrences of* D_1 *which have prefix-suffix conflicts with* D_2) $/f(D_1)$.

If $p(D_1, D_2)$ or $q(D_1, D_2)$ is greater than a statistically determined threshold, then the following could be be said with some confidence: *Presence of* D_1 *implies Presence of* D_2.

Let $psf(D_1, D_2, D_m)$ denote the number of prefix-suffix conflicts between D_1 and D_2 with respect to D_m and $psf(D_1, D_2)$, the number of prefix-suffix conflicts between D_1 and D_2.

We can approximate $p(D_1, D_2)$ by $rf(D_1, D_2) * f(D_2)/f(D_1)$. The two quantities are identical unless a single occurrence of D_1 is a subword of two or more distinct occurrences of D_2. Similarly, we can approximate $q(D_1, D_2)$ by $psf(D_1, D_2)/f(D_1)$. The two quantities are identical unless a single occurrence of D_1 has prefix-suffix conflicts with two or more distinct occurrences of D_2. $f(D_1)$ can be computed for all displayable entities in $CSD(S)$ in $O(n)$ time by a single traversal of $CSD(S)$ in

```
Procedure GetSubwords(S, v)
1     begin
2         rf(str(v), str(v)) = 1;
3         SetUp(v);
4         SetSuffixes(v);
5         for each vertex, x (≠ source), in reverse topological order of SG(S, v) do
6             begin
7                 if str(x) is a suffix of str(v) then rf(str(x), str(v)) = 1
8                 else rf(str(x), str(v)) = 0;
9                 for each w in SG(S, v) on which an re-edge e from x is incident do
10                        rf(str(x), str(v)):= rf(str(x), str(v)) + rf(str(w), str(v));
11                output(rf(str(x), str(v)));
12            end;
13    end
```

Figure 3.12. Modification to $GetSubwords(S, v)$ for computing relative frequencies

reverse topological order. $rf(D_1, D_2)$ may be computed optimally for all D_1, D_2, by modifying procedure $GetSubwords(S, v)$ as shown in Figure 3.12.

$psf(D_1, D_2, D_m)$ is computed optimally, for all D_1, D_2, and D_m, where D_1 has a prefix-suffix conflict with D_2 with respect to D_m, by modifying $ListConflicts(u, z, w)$ of Figure 3.9 so that it returns $f(str(u))$, since this is the number of conflicts between $str(w)$ and $str(z)$ with respect to $str(v)$. $psf(D_1, D_2)$ is calculated by summing $psf(D_1, D_2, D_m)$ over all intersections D_m of prefix-suffix conflicts between D_1 and D_2. $p(D_1, D_2)$ and $q(D_1, D_2)$ may be computed by simple modifications to the algorithms used to compute $rf(D_1, D_2)$ and $psf(D_1, D_2)$. These problems may be solved under the size restrictions of **P11** and **P12** by modifications similar to those made in Section 3.3.

3.6 Experimental Results

Algorithms *SubwordConflicts(S)* and *PrefixSuffixConflicts(S)* were implemented on a SUN SPARCstation 1 in GNU C++. 50 randomly generated strings of lengths ranging from 100 to 2000 from alphabets whose size ranged from 2 to 50 were input to our algorithms. Statistical information such as the number of vertices in the csdawg, the number of prefix-suffix and subword conflicts etc was obtained. The run times of the algorithms were also recorded.

(i) Figure 3.13 shows Number of prefix-suffix conflicts vs String size. There is one curve for each alphabet size. The plot illustrates that the number of prefix-suffix conflicts increases with string size and decreases with alphabet size. The graph for subword conflicts is similar.

(ii) Figure 3.14 shows Time per prefix-suffix conflict (μs) vs String size. There is one curve for each alphabet size. It illustrates that the time per conflict generally decreases with increasing string size and increases with alphabet size. The graph for subword conflicts is similar.

(iii) The factor by which the compact representation of subword conflicts is smaller than the fully expanded representation varies from 2 to 9. It increases with string size and decreases with alphabet size.

(iv) Table 3.1 shows the size of the largest displayable entity for each combination of alphabet size and string size. It shows that only displayable entities of small lengths occur in random strings (in practice we would like to be able to distinguish between displayable entities that occur randomly and those that do not in a given string. This can be done by selecting those displayable entities whose frequency is large compared to other displayable entities of the same length in a random string).

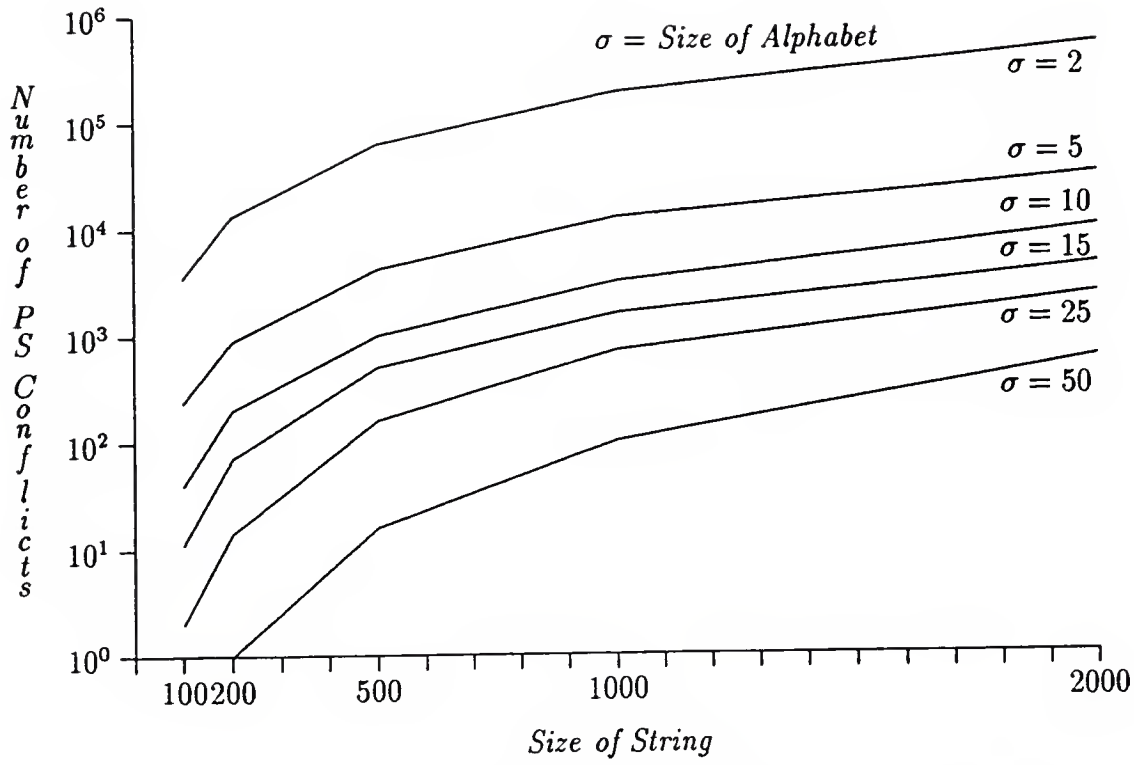


Figure 3.13. Graph of Number of Prefix-Suffix Conflicts vs String Size

Table 3.1. Lengths of Largest Displayable Entity for Random Strings

Size of	*Size of String*				
Alphabet	100	200	500	1000	2000
2	11	12	15	18	19
5	5	6	7	8	9
10	3	4	5	6	6
15	3	3	4	5	5
20	3	3	4	4	5
25	2	3	3	4	4
50	2	2	3	3	4

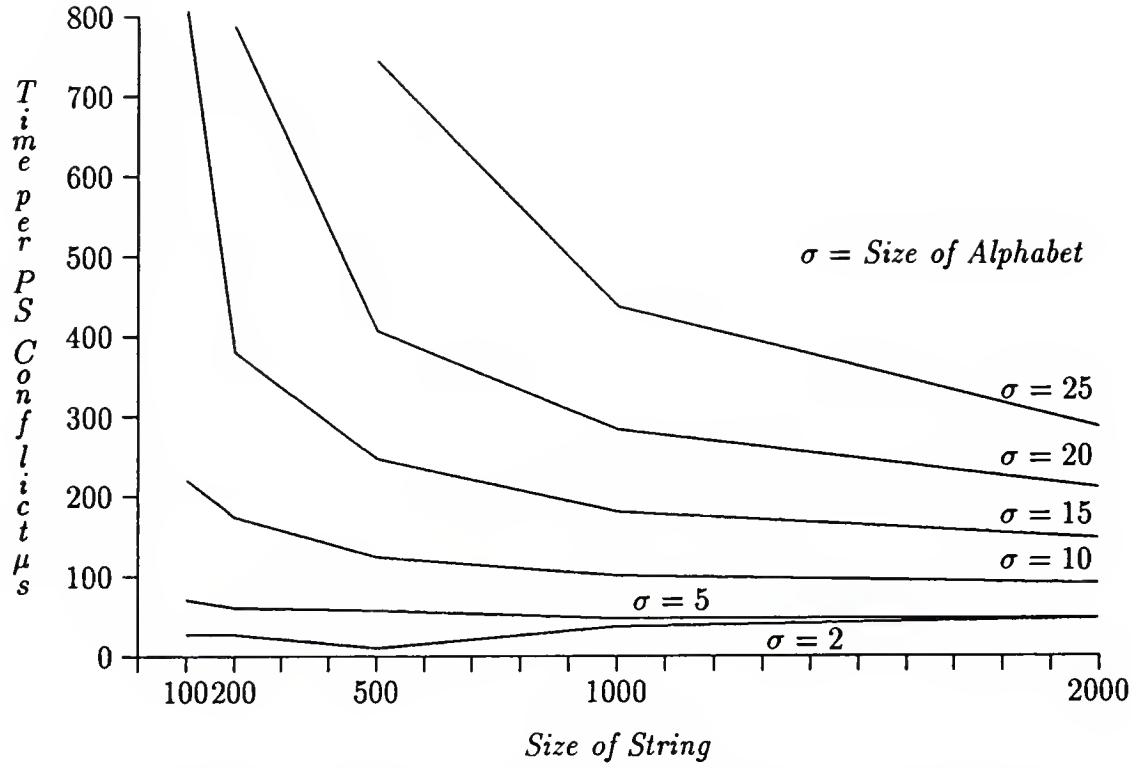


Figure 3.14. Graph of Time per Prefix-Suffix Conflict vs String Size

In another experimient, Algorithms *SubwordConflicts(S)* (Section 3.2.2), *PrefixSuffixConflicts(S)* (Section 3.2.3), and *AllConflicts(S)* (Section 3.2.4) were programmed in GNU C++ and run on a SUN SPARCstation 1. For test data we used 120 randomly generated strings. The alphabet size was chosen to be one of {5, 15, 25, 35} and the string length was 500, 1000, or 2000. The test set of strings consisted of 10 different strings for each of the 12 possible combinations of input size and alphabet size. For each of these combinations, the average run times for the 10 strings is given in Tables 3.2-3.5.

Table 3.2 gives the average times for computing all conflicts by combining algorithms *PrefixSuffixConflicts(S)* and *SubwordConflicts(S)*. Table 3.3 gives the average times for computing all prefix-suffix conflicts using **Algorithm** *PrefixSuffixConflicts(S)*. Table 3.4 gives the average times for computing all pattern-restricted prefix-suffix conflicts (problem **P14** of Section 3.4) by modifying **Algorithm** *PrefixSuffixConflicts(S)* as described in Section 3.4. Table 3.5 represents the average times for **Algorithm** *AllConflicts(S)*.

Tables 3.2 to 3.4 represent the theoretically superior solutions to the corresponding problems, while Table 3.5 represents **Algorithm** *AllConflicts(S)* which provides a simpler, but suboptimal, solution to the three problems. In all cases the time for constructing csdawgs and writing the results to a file were not included as these steps are common to all the solutions.

The results show that the suboptimal **Algorithm** *AllConflicts(S)* is superior to the optimal solution for computing all conflicts or all prefix-suffix conflicts for a randomly generated string. This is due to the simplicity of **Algorithm** *AllConflicts(S)* and the fact that the number of conflicts in a randomly generated string is small. However, on a string such as a^{100} which represents the worst case scenario in terms of the number of conflicts reported, the following run times were obtained:

Table 3.2. Time in ms for computing all conflicts using *SubwordConflicts(S)* and *PrefixSuffixConflicts(S)*

Size of Alphabet	*Size of String*		
	500	1000	2000
5	410	989	2722
15	292	603	1300
25	315	671	1485
35	234	791	1740

All conflicts, optimal algorithm: 14,190 ms

All prefix-suffix conflicts, optimal algorithm: 10,840 ms

All pattern restricted prefix-suffix conflicts, optimal algorithm: 5,000 ms

Algorithm *AllConflicts(S)*: 26,942 ms

The experimental results using random strings also show that, as expected, the optimal algorithm fares better than **Algorithm** *AllConflicts(S)* for the more restricted problem of computing pattern oriented prefix-suffix conflicts.

We conclude that **Algorithm** *AllConflicts(S)* should be used for the more general problems of computing conflicts while the optimal solutions should be used for the restricted versions. Hence, **Algorithm** *AllConflicts(S)* should be used in an automatic environment, while the optimal solutions should be used in interactive or semi-automatic environments.

3.7 Display Algorithms

A list of all occurrences of a displayable entity may be obtained from the csdawg data structure described earlier. A list of all conflicts between displayable entities may be obtained by operations on the csdawg. This information is then used to assign numeric weights to each occurrence of each displayable entity.

Table 3.3. Time in ms for computing all prefix-suffix conflicts using *PrefixSuffixConflicts(S)*

Size of Alphabet	*Size of String*		
	500	1000	2000
5	247	730	1873
15	219	454	989
25	255	522	1179
35	186	648	1370

Table 3.4. Time in ms for computing all pattern restricted prefix-suffix conflicts using the optimal algorithm

Size of Alphabet	*Size of String*		
	500	1000	2000
5	163	399	1058
15	103	231	550
25	91	267	628
35	61	226	735

Table 3.5. Time in ms for Algorithm *AllConflicts(S)*

Size of Alphabet	*Size of String*		
	500	1000	2000
5	203	551	1367
15	217	409	897
25	227	400	887
35	145	478	994

In this section we shall discuss algorithms to implement some of the refined display models of Section 2.3. Specifically, we shall consider models 1, 2(b), and 3 under the Automatic mode (i.e., problems **P4**, **P6**, and **P7** of Section 2.4). Our algorithms also apply to the Semi Automatic mode. Problem **P4** can be reduced to the single pair, longest path problem in a directed acyclic graph as follows: let vertex V_i, $1 \leq i < n$, correspond to the position between the i'th and $i+1$'th characters in the string. V_0 corresponds to the position preceding the first character in the string. V_n corresponds to the position following the last character in the string. For each occurrence $S_{i,j}$ of each displayable entity, we create an edge from V_{i-1} to V_j. The weight associated with this edge is exactly the weight of the occurrence it represents. Finally, for each pair (V_i, V_{i+1}), $0 \leq i < n$, of vertices such that an edge from V_i to V_{i+1} does not already exist, we create an edge from V_i to V_{i+1} of weight 0.

Figure 3.15 shows the directed acyclic graph corresponding to the string $S =$ *abcicdefcdegabchancde* of Figure 2.2, assuming that the weights corresponding to each occurrence of *abc*, *cde*, and *c* are 4, 3, and 2, respectively. The longest path from V_0 to V_n in the dag is $V_0 \rightarrow V_3 \rightarrow V_4 \rightarrow V_7 \rightarrow V_8 \rightarrow V_{11} \rightarrow V_{12} \rightarrow V_{15} \rightarrow V_{16} \rightarrow V_{19} \rightarrow V_{20} \rightarrow V_{21}$. All edges on this path with non-zero weight represent occurrences of displayable entities that are to be highlighted. Here $V_0 \rightarrow V_3$, $V_4 \rightarrow V_7$, $V_8 \rightarrow V_{11}$, $V_{12} \rightarrow V_{15}$, $V_{16} \rightarrow V_{19}$ correspond to occurrences $< 1,3 >$, $< 13,15 >$, and $< 17,19 >$ of *abc*, and $< 5,7 >$ and $< 9,11 >$ of *cde*. The length of the longest path (here, 18) represents the total weight of the display.

Algorithm *A4* of Figure 3.16 solves problem **P4**. $A[0..n]$ is an array of integers, where $A[i]$ represents the longest path from V_0 to V_i detected upto that point of time. All elements of A are initialized to 0. The auxiliary array $T[1..n]$ stores the occurrences that have been chosen for display. The array *de_list* contains all the occurrences of all the displayable entities in the string. Each element of *de_list*

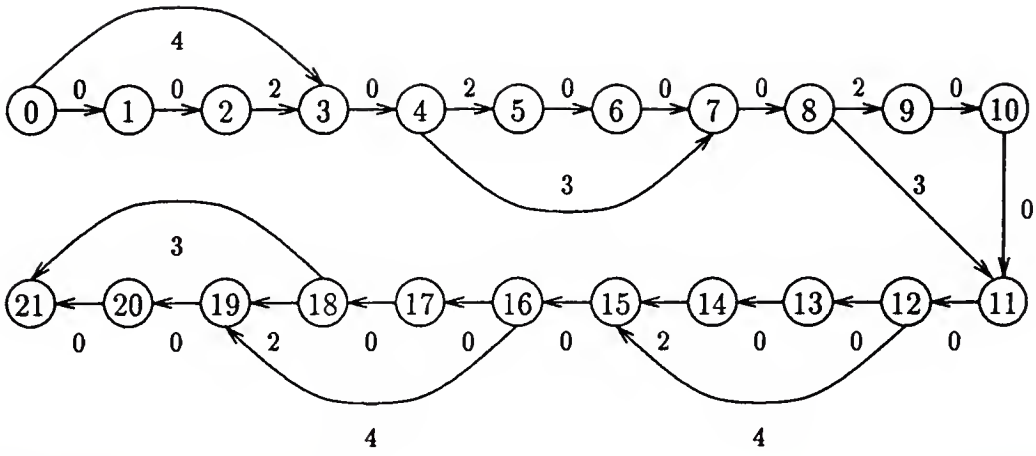


Figure 3.15. Dag corresponding to *abcicdefcdegabchabcde*

contains three fields: *start*, *end*, and *weight* which represent the start position, the end position, and the numeric weight of the occurrence, respectively. It is assumed that *de_list* is sorted in increasing order of *end*. The vertices are processed in topological order (here, V_0, V_1, ..., V_n). When a vertex V_j is being processed, each vertex V_i ($i < j$) preceding it has associated with it the cost of the longest path from V_0 to V_i. The cost of the longest path from V_0 to V_j is then determined by examining each of the incoming edges to V_j.

The complexity of **Algorithm** *A4* is $O(n + e)$, where e represents the size of *de_list*. So, **Algorithm** *A4* is optimal. Note that sorting *de_list* on end position can also be accomplished in $O(n + e)$ time using radix sort.

Problem **P6** of Section 2.4 is solved by **Algorithm** *A6* of Figure 3.17 using the greedy method. The array *endpoints[1..2e]* is a list of, both, the start positions and end positions of all occurrences of all displayable entities in the string. Each element of *endpoints* contains three fields: *position*, *type*, and *id*. *position* contains the position of

```
Algorithm A4
begin
   A[0]:= 0; j:= 1;
   for i := 1 to n do
      begin
         A[i]:= 0; {Initialize A[i]}
         while (de_list[j].end = i) do {determine longest path ending at V_i}
            begin
               if A[i] < A[de_list[j].start-1] + de_list[j].weight
               then
                  begin
                     A[i] := A[de_list[j].start-1] + de_list[j].weight;
                     T[i] := j;
                  end;
               j:= j + 1;
            end;
      end;
end.
```

Figure 3.16. Algorithm for **P4**

the particular endpoint in the string; *type* indicates whether the endpoint is a "start" position or an "end" position. *id* is an integer which uniquely identifies the occurrence corresponding to the endpoint. It is assumed that *endpoints* is sorted in increasing order on primary key *position* and secondary key *type* ("start" < "end"). The variable *current* keeps track of the number of copies of the string that are currently "active." The variable *max* keeps track of the maximum number of copies of the string required so far. *CurrentLine* is the particular copy of the string to which the new occurrence is assigned. *LineStack* is a stack that contains line numbers (or partition numbers) which are currently available. On completion of the algorithm, *line*[*i*] contains the line or the particular copy of the string in which the occurrence with *id i* is to be highlighted for $1 \leq i \leq e$.

It can be seen from the algorithm that

(1) At least one position in the string is covered by *fmax* occurrences. Thus, *fmax* is the smallest number of lines required to highlight all the displayable entities.

(2) The final value of *max* represents the number of partitions of S required to highlight all displayable entities. Since the the final value of *max* is *fmax*, **Algorithm** *A6* is correct.

Algorithm *A6* consumes $O(n + e)$ time, which is optimal. Note that sorting *endpoints* can also be accomplished in $O(n + e)$ time, if radix sort is used.

We outline two solutions to **P7**. In the first, we assume that all occurrences of the same displayable entity are assigned the same numeric weight. In the second, we do not make this assumption.

Algorithm *A7(a)* of Figure 3.18 solves the first version of **P7**. The second version of **P7** may be solved by executing steps a-c for each occurrence of each displayable entity as shown in **Algorithm** *A7(b)* of Figure 3.19. Both solutions involve a traversal of $CSD(S)$ in topological order. For each occurrence, an optimal

```
Algorithm A6
begin
   max := 0; current := 0;
   for i := 1 to 2e do
      begin
         x := endpoints[i];
         if (x.type = "start") then
            begin {assign line numbers to occurrences}
               current := current + 1;
               if (current ≤ max) then {use available copy of string}
                  begin
                     CurrentLine := top(LineStack);
                     pop(LineStack);
                  end
               else
                  begin {increase max }
                     max := max + 1;
                     CurrentLine := max;
                  end;
               line[x.id] := CurrentLine;
            end
         else {x.type = "end"}
            begin
               current := current - 1;
               push(LineStack, line[x.id]);
            end;
      end;
   fmax := max;
end.
```

Figure 3.17. Algorithm for **P6**

Algorithm *A7(a)*
begin
for each vertex v of $CSD(S)$ in topological order **do**
begin
Step a. Compute the relative positions of all subword displayable entities in a single instance of $str(v)$ using **Procedure** *GetSubwords(S,v).*
Step b. Choose a mutually non overlapping subset from the set of subwords of $str(v)$ (obtained in step (a) above) so that the sum of their weights is maximized. This is achieved by an algorithm similar to A4.
Step c. Reset the numeric weight of $str(v)$ by adding to it the total weight of the configuration obtained in step (b).
end;
end.

Figure 3.18. Algorithm for **P7**, same weights

selection of its subwords are chosen. The weight of the occurrence is then obtained by adding the sum of the weights of the chosen subwords to its weight. This is done because an occurrence is highlighted along with the chosen subwords. **Algorithm** *A7(a)* consumes $O(n^3)$ time, while **Algorithm** *A7(b)* consumes $O(n^4)$ time.

Algorithm *A7(b)*
begin
for each vertex v of $CSD(S)$ in topological order **do**
for each occurrence $< i, j >$ of $str(v)$ **do**
begin
Step a. Compute the occurrences of all subword displayable entities in $< i, j >$.
Step b. Choose a mutually non overlapping subset from the set of occurrences (obtained in Step (a) above) so that the sum of their weights is maximized. This is achieved by an algorithm similar to A4.
Step c Reset the numeric weight of $< i, j >$ by adding to it the total weight of the configuration obtained in Step (b).
end;
end.

Figure 3.19. Algorithm for **P7**, different weights

CHAPTER 4
CIRCULAR STRING VISUALIZATION

4.1 Introduction

The circular string data type is used to represent a number of objects such as circular genomes, polygons, and closed curves. Research in molecular biology involves the identification of recurring patterns in data and hypothesizing about their causes and/or effects [4, 2]. Research in pattern recognition and computer vision involves detecting similarities within an object or between objects [11]. We have already listed in Chapter 2 a number of queries that our visualization model supports. In Chapter 3, we developed efficient (mostly optimal) algorithms for some of these queries for linear strings. These algorithms performed operations and traversals on csdawgs of the linear strings.

One approach for extending these techniques to circular strings is to arbitrarily break the circular string at some point so that it becomes a linear string. Techniques for linear strings may then be applied to it. However, this has the disadvantage that some significant patterns in the circular string may be lost because the patterns were broken when linearizing the string. Indeed, this would defeat the purpose of representing objects by circular strings. This particular problem may be overcome by working with the csdawg corresponding to the concatenation of the linearized circular string with itself. However, the resulting data structure contains a number of extraneous nodes. The existence of these nodes increases the asymptotic complexity

and actual running times of some of our algorithms, and increases the storage requirement of the data structure. Moreover, algorithms for linear strings need to be substantially modified for use with this data structure.

A circular string data structure, the polygon structure graph, which is an extension of suffix trees to circular strings already exists [11]. However, the suffix tree is not as powerful as the csdawg and cannot be used to solve some of the problems that the csdawg can solve. In particular, our queries required the csdawg for asymptotically and practically efficient algorithms.

In this chapter, we propose a csdawg for circular strings which is obtained by making simple modifications to the csdawg for linear strings. This new data structure does not contain extraneous vertices and consequently avoids the disadvantages mentioned above. Algorithms which make use of the csdawg for linear strings can then be extended to circular strings with trivial modifications. The extended algorithms continue to have the same time and space complexities. Moreover, the extensions take the form of postprocessing or preprocessing steps which are simple to add on to a system built for linear strings, particularly in an object-oriented language. In particular, algorithms *NoConflicts(S)*, *SubwordConflicts(S)*, *PrefixSuffixConflicts(S)*, *AllConflicts(S)* and the solutions outlined for problems **P8** to **P18** in Chapter 3 for linear strings can be easily extended to circular strings.

Section 4.2 contains definitions. Section 4.3 describes the construction of a circular csdawg and the computation of locations of occurrences of displayable entities and Section 4.4 introduces the notion of display conflicts. Finally, Section 4.5 mentions some applications for the visualization and analysis of circular strings.

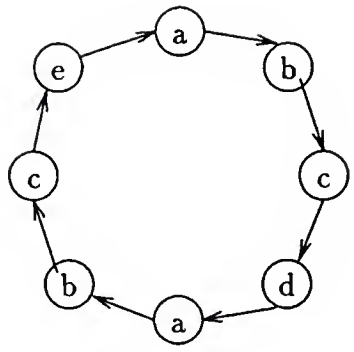


Figure 4.1. Circular string

4.2 Definitions

Let s denote a circular string of size n consisting of characters from a fixed alphabet Σ of constant size. Figure 4.1 shows an example circular string of size 8. We shall represent a circular string by a linear string enclosed in angle brackets "<>" (this distinguishes it from a linear string) . The linear string is obtained by traversing the circular string in clockwise order and listing each element as it is traversed. The start point of the traversal is chosen arbitrarily. Consequently, there are up to n equivalent representations of s. In the example, s could be represented as <*abcdabce*>, <*bcdabcea*>, etc.

We characterize the relationship between circular strings and linear strings by defining the functions *linearize* and *circularize*. *linearize* maps circular strings to linear strings. It is a one-many mapping as a circular string can, in general, be mapped to more than one linear string. For example, *linearize*(<*abcd*>) = {*abcd*, *bcda*, *cdab*, *dabc*}. We will assume that *linearize* arbitrarily chooses one of the linear strings; for convenience we assume that it chooses the representation obtained by removing the angle brackets "<>." So, *linearize*(<*abcd*>) = *abcd*. *circularize* maps

linear strings to circular strings. It is a many-one function and represents the inverse of *linearize*.

We use lower case letters to represent circular strings and upper case letters to represent linear strings. Further, if a lower case letter (say s) is used to represent a particular circular string, then the corresponding upper case letter (S) is assumed to be *linearize*(s).

The definitions of maximal strings and displayable entities for circular strings are identical to those for linear strings. I.e., a non-null pattern occurring in s is said to be *maximal* iff its occurrences are not all preceded by the same character, nor all followed by the same character. The empty string is always maximal. s may be thought of as a periodic, infinite string which is neither followed nor preceded by a letter, and is therefore maximal. A pattern is said to be a *displayable entity* of s iff it is non-null, maximal, and occurs at least twice in σ. A displayable entity of s always has length less than n. Thus, s and the empty string are maximal, but not displayable entities.

The definition of a csdawg for circular strings is also similar to that for linear strings. I.e., a csdawg, $CSD(s) = (V(s), R(s), L(s))$ corresponding to s is a directed acyclic graph defined by a set of vertices $V(s)$, a set $R(s)$ of labeled directed edges called right extension edges (re-edges), and a set of labeled directed edges $L(s)$ called left extension edges (le-edges). Each vertex of $V(s)$ represents a substring of s. Specifically, $V(s)$ consists of a vertex corresponding to each maximal pattern of s. This consists of a source which represents the empty string λ; a sink which represents s; and a vertex for each displayable entity of s.

Let $str(v)$ denote the string represented by vertex v for $v \epsilon V(s)$. Define the *implication* $imp(s, \alpha)$ of a string α in s to be the smallest superstring of α in $\{str(v) | v \epsilon V(s)\}$, if such a superstring exists. Otherwise, $imp(s, \alpha)$ does not exist. Note that

$imp(s, \alpha)$ is always unique (if there are two or more superstrings of α of the same length k in $\{str(v)|\ v \in V(s)\}$, it can be shown that there must exist a superstring of α in $\{str(v)|\ v \in V(s)\}$ with length less than k). Re-edges from v_1 ($v_1 \in V(s)$) are obtained as follows: for each letter x in Σ, if $imp(s, str(v_1)x)$ exists and is equal to $str(v_2) = \beta str(v_1)x\gamma$, then there is an re-edge from v_1 to v_2 with label $x\gamma$. If β is the empty string, then the edge is known as a *prefix extension edge*. Le-edges from v_1 ($v_1 \in V(s)$) are obtained as follows: for each letter x in Σ, if $imp(s, xstr(v_1))$ exists and is equal to $str(v_2) = \gamma x str(v_1)\beta$, then there is an le-edge from v_1 to v_2 with label γx. If β is the empty string, then the edge is known as a *suffix extension edge*

The sink represents the periodic infinite string denoted by the circular string s. The labels of edges incident on the sink are themselves infinite and periodic and may be represented by their start positions in s. We also associate with $CSD(s)$ the *periodicity* p of s. This is the value of $|\alpha|$ for the largest value of k such that $S = \alpha^k$. We require the periodicity to answer queries about the locations of displayable entities if S is itself periodic. Figure 4.11 shows $CSD(s)$ for $s = <cabcbab>$. Its periodicity is 7.

4.3 Constructing the Csdawg for a Circular String

The csdawg for circular string s is constructed by the algorithm of Figure 4.2. It is obtained by first constructing the csdawg for the linear string $T = SS$ (recall that $S = linearize(s)$). A bit is associated with each re-edge in $R(T)$ indicating whether it is a prefix extension edge or not. Similarly, a bit is associated with each le-edge in $L(T)$ to identify suffix extension edges. Two pointers, a suffix pointer and a prefix pointer are associated with each vertex v in $V(T)$. The suffix (prefix) pointer points to a vertex w in $V(T)$ such that $str(w)$ is the largest proper suffix (prefix) of $str(v)$ represented by any vertex in $V(T)$. Note that such a w always exists since the empty

Algorithm *CircularCsdawg(s)*
Step1: Construct $CSD(T)$ for $T = SS$ (S = *linearize(s)*).

Step2:
{Determine periodicity of S using Lemma 4.3.1}
v_S := vertex representing S in $CSD(T)$
e_S := any outgoing edge from v_S
$p := |e_S|$

Step3(a):
{Identify Suffix Redundant Vertices using Lemma 4.3.3}
v:= *sink*;
while $v \neq$ *source* **do**
  begin
    v:= *v.suffix*;
    if v has exactly one outgoing re-edge **then** mark v suffix redundant;
    else exit Step 3(a);
  end;

Step3(b):
{Identify Prefix Redundant vertices}
{Similar to Step 3(a)}

Step4:
v:= *sink*;
while (v <> *source*) **do**
  begin
    Modify representation of edges from v that are incident on the sink.
    case v **of**
      suffix redundant but not prefix redundant: *ProcessSuffixRedundant(v)*;
      prefix redundant but not suffix redundant: *ProcessPrefixRedundant(v)*;
      suffix redundant and prefix redundant : *ProcessBothRedundant(v)*;
      not redundant: {Do nothing};
    endcase;
    v:= *NextVertexInReverseTopologicalOrder*;
  end;

Figure 4.2. Algorithm for constructing the csdawg for a circular string

string is a suffix (prefix) of all strings. Suffix (prefix) pointers are the reverse of suffix (prefix) extension edges and are derived from them. Figure 4.4 shows $CSD(T) = CSD(SS)$ for $S = cabcbab$. The broken edge from vertex c to vertex abc is a suffix extension edge, while the solid edge from vertex ab to vertex abc is a prefix extension edge.

In Step 2, we determine the periodicity of s which is equal to the length of the label on any outgoing edge from the vertex representing S in $CSD(T)$. This equality is derived in Lemma 4.3.1.

Lemma 4.3.1 *Step 2 of* Algorithm *CircularCsdawg(s) correctly determines the periodicity of s.*

Proof: Let α be the shortest substring of S such that $S = \alpha^m$, for some integer m. Then, the occurrences of α in T have start positions 1, $|\alpha|+1$,$2|\alpha|+1$,...,$(2m-1)\alpha+1$ (if this is not the case, then we can show that there exists a substring β of S such that $S = \beta^k$, $k > m$, resulting in a contradiction). So, $CSD(T)$ takes the form of Figure 4.3. Each vertex representing α^i, $1 \leq i \leq 2m-1$, has exactly one le-edge and one re-edge leaving it as shown. All remaining displayable entities of T are subwords of α^2 and are of size less than $|\alpha|$ (if this is not the case, then we can show that there exists a substring β of S such that $S = \beta^k$, $k > m$, resulting in a contradiction). The vertices representing these displayable entities are represented by the box in Figure 4.3. From the figure, it is now easy to see that Step 2 correctly determines the periodicity of s. ■

Next, in Step 3, suffix and prefix redundant vertices of $CSD(T)$ are identified. A *suffix (prefix) redundant* vertex is one that has exactly one re-edge (le-edge) leaving it. A vertex is said to be *redundant* if it is either prefix redundant or suffix redundant

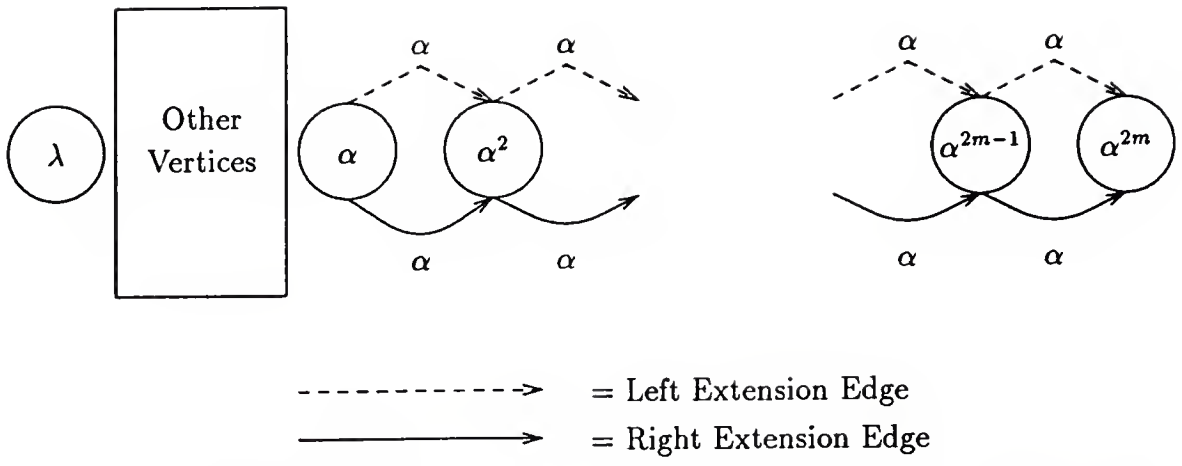


Figure 4.3. $SCD(\alpha^{2m})$

or both. We have, from Lemma 4.3.2, that redundant vertices in $CSD(T)$ are the only vertices in $CSD(T)$ (except the source and the sink) which do not represent displayable entities of s. Redundant vertices represent patterns that are maximal in T specifically because they occur at either end of T and therefore are not followed or preceded by a letter.

Lemma 4.3.2 *A vertex v in $V(T)$ ($v \neq$ source, $v \neq$ sink) is not redundant iff $str(v)$ is a displayable entity of s.*

Proof: A displayable entity of s must be a displayable entity of T since, by construction of T, any pattern of size less than n in s has at least once occurrence in T that is preceded (followed) by any letter that preceded (followed) an occurrence of the same pattern in s. Further, a displayable entity of s must have at least two occurrences which are preceded by different letters and at least two occurrences which are followed by different letters (note that a displayable entity of a circular string, unlike that of a linear string, is always preceded and followed by a letter). So, from

the definition of a csdawg, a vertex in $V(T)$ corresponding to a displayable entity of s must have two re-edges and two le-edges and is not redundant.

All displayable entities of T of size $\geq n$ are redundant (from Figure 4.3). All non-redundant vertices of T have at least two le-edges and two re-edges. Since they represent strings of size less than n, and since, by construction, any string of size less than n in T is a string in s, they must represent displayable entities of s.

■

In Figure 4.4, vertex c is prefix redundant only, while vertex ab is suffix redundant only. The vertex representing S is both prefix and suffix redundant since it has one re-edge and one le-edge leaving it. The fact that Step 3 does, in fact, identify all redundant vertices is established by Lemma 4.3.3.

Lemma 4.3.3 *(a) A vertex v in $V(T)$ will have exactly one re-edge (le-edge) leaving it only if $str(v)$ is a suffix (prefix) of T.*
(b) If a vertex v such that $str(v)$ is a suffix (prefix) of T has more than one re-edge (le-edge) leaving it, then no vertex w such that $str(w)$ is a suffix (prefix) of $str(v)$ can be suffix (prefix) redundant.

Proof: (a) Suppose $str(v)$ is not a suffix of T. Then v has at least two re-edges leaving it, otherwise $str(v)$ would not be maximal in T. But this is a contradiction and $str(v)$ must be a suffix of T.
(b) Since $str(w)$ is a suffix of $str(v)$, a letter following $str(v)$ must also follow $str(w)$. So, w must have at least as many re-edges leaving it as v. But v has at least two re-edges leaving it, so w cannot be suffix redundant ■

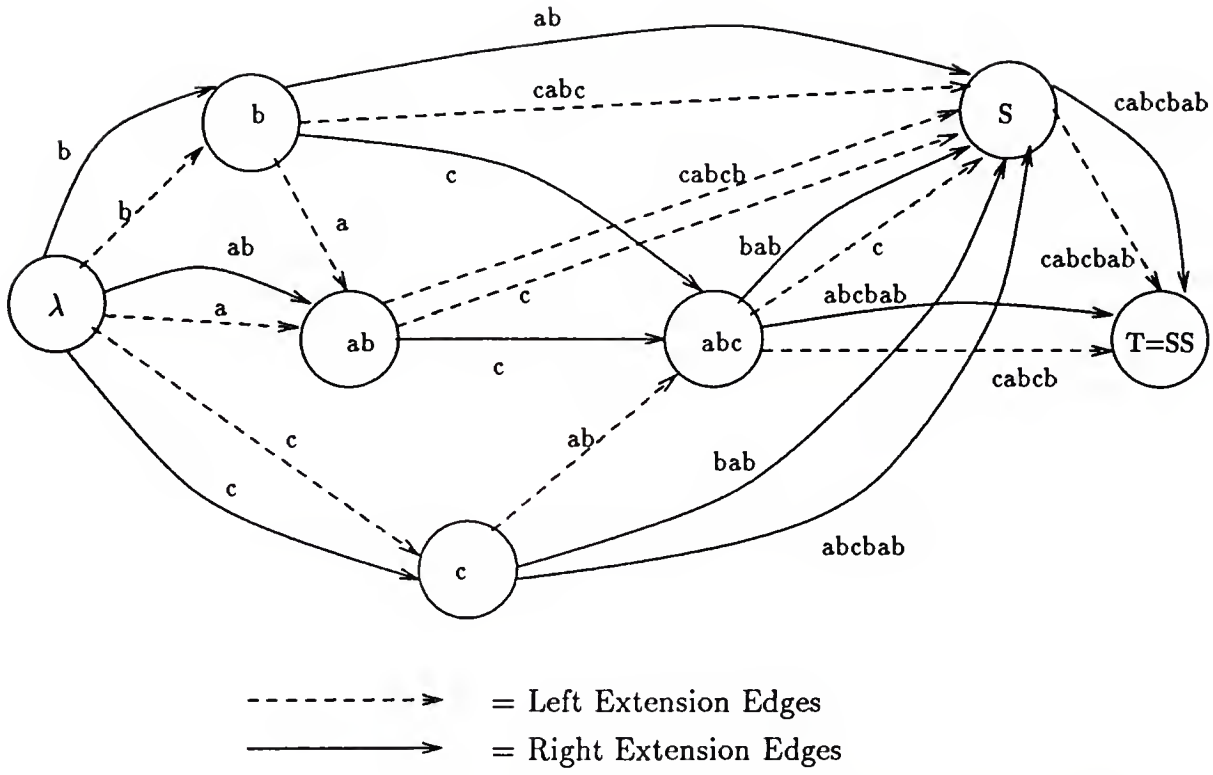


Figure 4.4. $CSD(T)$ for T=*cabcbabcabcbab*

Since it is sufficient to examine vertices corresponding to suffixes of T (Lemma 4.3.3(a)), Step 3(a) follows the chain of suffix pointers starting from the sink. If a vertex on this chain has one re-edge leaving it, then it is marked suffix redundant. The traversal of the chain terminates either when the source is reached or when a vertex with more than one re-edge leaving it is encountered (Lemma 4.3.3(b)). Similarly, Step 3(b) identifies all prefix redundant vertices in $V(T)$.

Vertices of $CSD(T)$ are processed in reverse topological order in Step 4 and redundant vertices are eliminated. When a vertex is eliminated, the edges incident to/from

it are redirected and relabeled as described in Figures 4.5 to 4.10. **Procedure** *ProcessPrefixRedundant* is symmetric to *ProcessSuffixRedundant.* The correctness of the relabeling and redirecting of edges in *ProcessSuffixRedundant* and *ProcessBothRedundant* follows from Lemmas 4.3.4 and 4.3.5. The resulting graph is $CSD(s)$.

Lemma 4.3.4 *In* **Procedure** *ProcessSuffixRedundant,* $str(v)$ *is a prefix of* $str(w)$.

Proof: All occurrences of $str(v)$ in s are followed by $x\gamma$. So, $str(w)$ must be of the form $\beta str(v)x\gamma$ since w cannot be redundant (otherwise it would have been eliminated in Step 4). But, at least two occurrences of *str(v)* are preceded by different letters (since v is not prefix-redundant). So, at least two occurrences of $str(v)x\gamma$ are preceded by different letters. So, $\beta = nil$, and $str(w) = str(v)x\gamma$. ■

Lemma 4.3.5 *In* **Procedure** *ProcessBothRedundant,* $w_1 = w_2$.

Proof: βy precedes all occurrences of $str(v)$ and $x\gamma$ follows all occurrences of $str(v)$ in s. Since w_1 and w_2 cannot be redundant (otherwise they would have been eliminated in Step 4), $str(w_1)$ and $str(w_2)$ are both $\beta y str(v)x\gamma$. ■

Procedure *ProcessSuffixRedundant*(v)

1. Eliminate all left extension edges leaving v (there are at least two of these).
2. There is exactly one right extension edge e leaving v. Let the vertex that it leads to be w. Let the label on the right extension edge be $x\gamma$. Delete the edge.
3. All right edges incident on v are updated so that they point to w. Their labels are modified so that they represent the concatenation of their original labels with $x\gamma$.
4. All left edges incident on v are updated so that they point to w. Their labels are not modified. However, if any of these were suffix extension edges, the bit which indicates this should be reset as these edges are no longer suffix extension edges.
5. Delete v.

Figure 4.5. Algorithm for processing a vertex which is suffix redundant

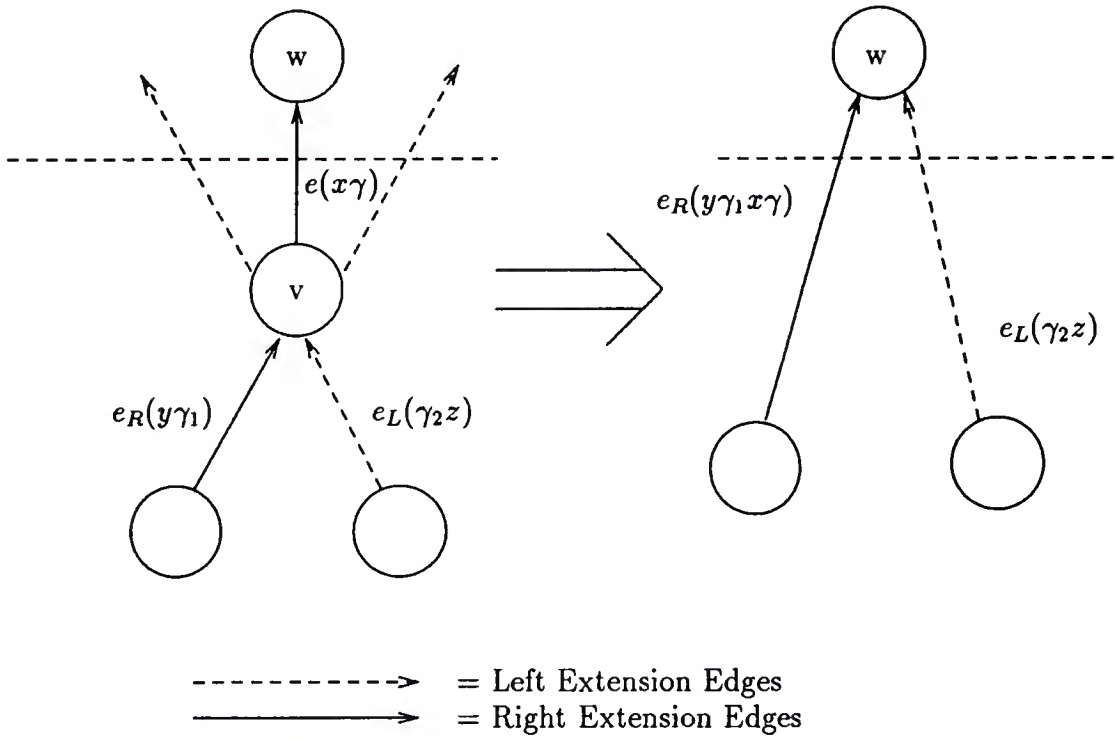


Figure 4.6. v is suffix redundant

Procedure *ProcessPrefixRedundant*(v)

1. Eliminate all right extension edges leaving v (there are at least two of these).
2. There is exactly one left extension edge e leaving v. Let the vertex that it leads to be w. Let the label on the left extension edge be γx. Delete the edge.
3. All left edges incident on v are updated so that they point to w. Their labels are modified so that they represent the concatenation of their original labels with γx.
4. All right edges incident on v are updated so that they point to w. Their labels are not modified. However, if any of these were prefix extension edges, the bit which indicates this should be reset as these edges are no longer prefix extension edges.
5. Delete v.

Figure 4.7. Algorithm for processing a vertex which is prefix redundant

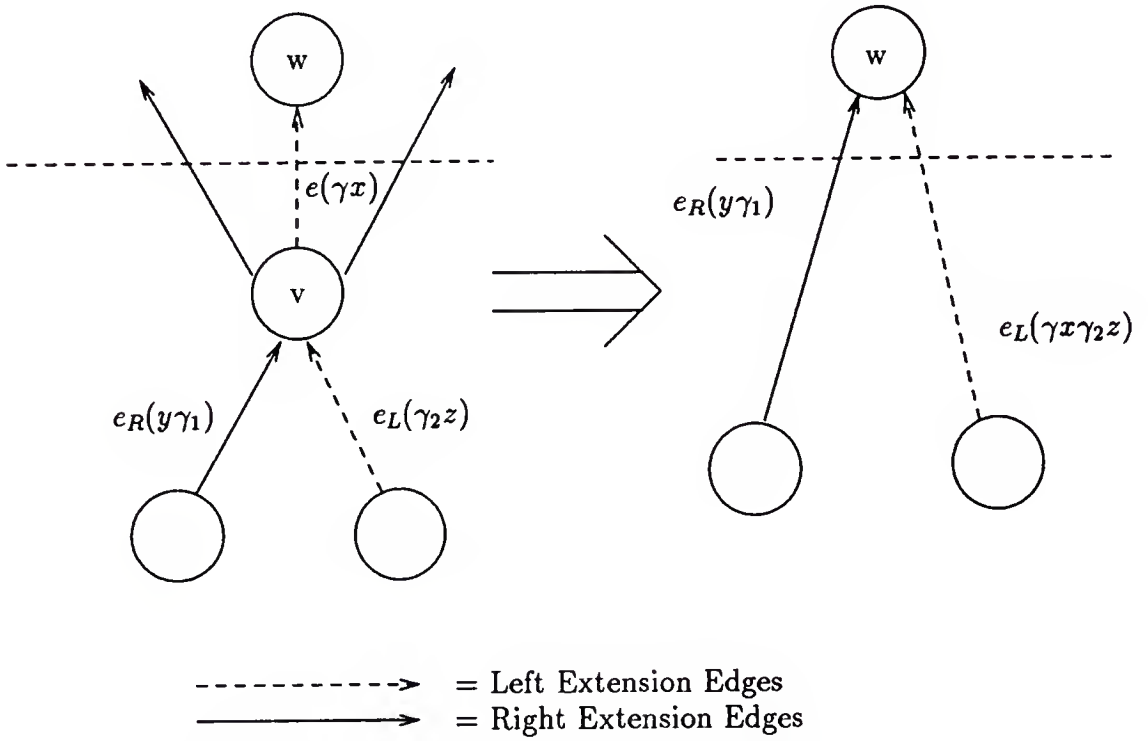


Figure 4.8. v is prefix redundant

Procedure *ProcessBothRedundant*(v)

1. There is exactly one right extension edge e_1 leaving v. Let the vertex that it leads to be w_1. Let the label on the edge be $x\gamma$. Delete the edge.
2. There is exactly one left extension edge e_2 leaving v. Let the vertex that it leads to be w_2. Let the label on the edge be βy. Delete the edge.
   {Lemma 4.3.5 establishes that w_1 and w_2 are the same vertex.}
3. All right edges incident on v are updated so that they point to w_1. Their labels are modified so that they represent the concatenation with $x\gamma$. If any of these edges were prefix edges, the bit which indicates this should be reset.
4. Similarly, left edges incident on v are updated so that they point to w_2. Their labels are modified so that they represent the concatenation with γx. If any of these edges were suffix extension edges, the bit which indicates this should be reset.
5. Delete v.

Figure 4.9. Algorithm for processing a vertex which is prefix and suffix redundant

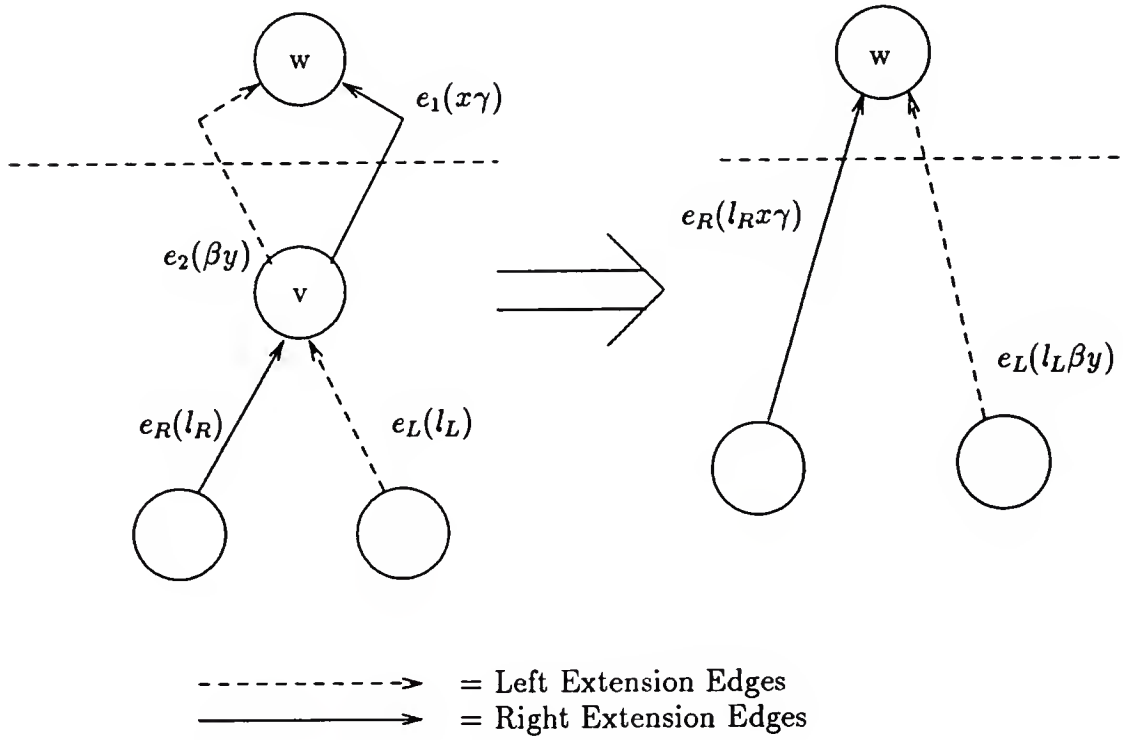


Figure 4.10. v is suffix and prefix redundant

Theorem 4.3.1 **Algorithm** *CircularCsdawg(s) correctly computes CSD(s) and has complexity O(n), which is optimal.*

Proof: Since the algorithm eliminates redundant vertices, Lemma 4.3.2 ensures that the vertices in $CSD(s)$ are correctly obtained. It remains to show that the edges of $CSD(s)$ are correctly obtained. Given Lemmas 4.3.4 and 4.3.5, it is easy to verify that Procedures *ProcessSuffixRedundant, ProcessPrefixRedundant*, and *ProcessBothRedundant* correctly relabel and redirect edges in the csdawg.

Step 1 takes $O(n)$ time [9]. Step 2 takes $O(n)$ time to locate the vertex representing S in $CSD(T)$. Step 3 will, in the worst case, traverse all the vertices in $CSD(T)$ spending $O(1)$ time at each vertex. The number of vertices is bounded by $O(n)$ [9]. So, Step 3 takes $O(n)$ time. Step 4 traverses $CSD(T)$. Each vertex is processed once; each edge is processed at most twice (once when it is an incoming edge to the vertex currently being processed, and once when it is the out edge from the vertex currently being processed). So, Step 4 takes $O(n)$ time (note that $CSD(T)$ has $O(n)$ edges). ■

Procedure *CircOccurrences*(s, v) of Figure 4.12 reports the end position of each occurrence of $str(v)$, for $v \in V(s)$, in the circular string s. The label corresponding to an edge terminating at a vertex other than the sink is denoted by *label(e)*. The start position of an edge terminating at the sink vertex is denoted by *pos(e)*. It is similar to the algorithm for computing occurrences of displayable entities in linear strings (Chapter 3).

4.4 Computing Conflicts Efficiently

We have identified a number of problems relating to the computation of conflicts in a linear string and have presented efficient algorithms for most of these problems

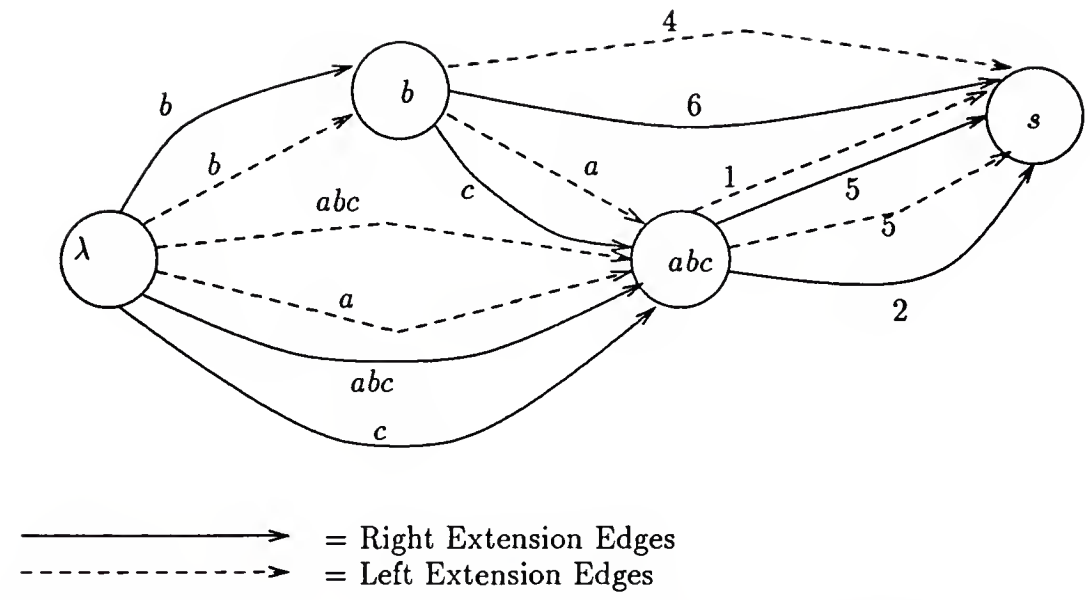


Figure 4.11. Csdawg for $< cabcbab >$

(Chapter 3). These algorithms typically involved sophisticated traversals or operations on the csdawg for linear strings. Our extension of csdawgs to circular strings makes it possible to use the same algorithms to solve the corresponding problems for circular strings with some minor modifications caused by our representation of edges that are incident on the sink.

4.5 Applications

Circular strings may be used to represent circular genomes [4] such as $G4$ and $\phi X174$. The detection and analysis of patterns in genomes helps to provide insights into the evolution, structure, and function of organisms. Here [4], $G4$ and $\phi X174$ are analyzed by linearizing them and then constructing their csdawg. We improve upon this by (i) analyzing circular strings without risking the "loss" of patterns. (ii)

```
Procedure CircOccurrences(s:circular string, v:vertex)
{Obtain all occurrences of str(v), v ∈ V(s), in S}
   if v ≠ sink then Occurrences(s,v,0);


Procedure Occurrences(s:circular string, v:vertex, i:integer)
   begin
      for each re-edge e from v in CSD(s) do
         begin
            let w be the vertex on which e is incident;
            if w ≠ sink then Occurrences(s,w,|label(e)| + i)
            else
               for k := 1 to (|s|/periodicity(s)) do
                  output(pos(e)-i-1+(k-1)periodicity(s))
         end;
   end;
```

Figure 4.12. Obtaining all occurrences of a displayable entity in a circular string

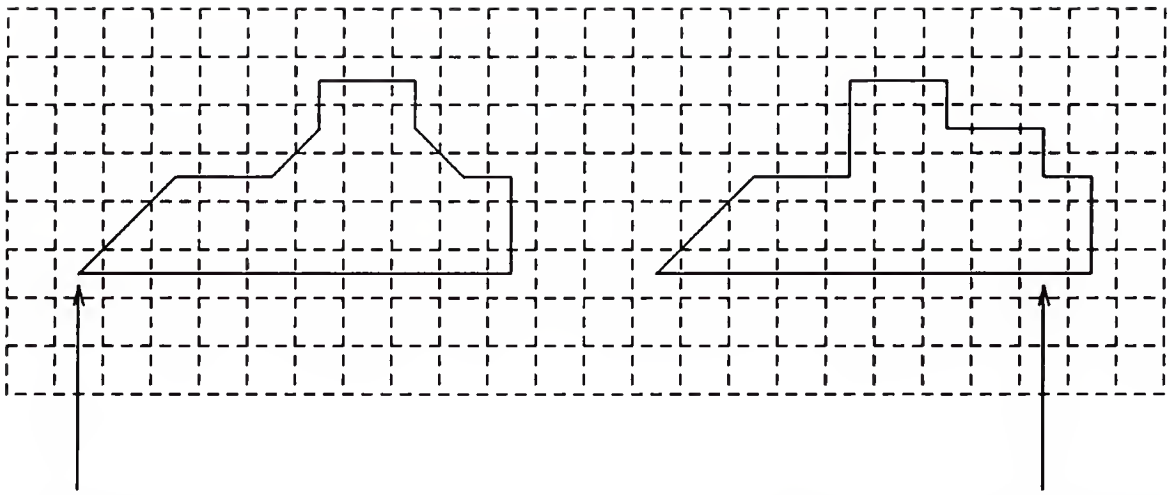
extending the analysis and visualization techniques of Chapter 3 for linear strings to circular strings.

Circular strings in the form of chain codes are also used to represent closed curves in computer vision [12]. The objects of Figure 4.13(a) are represented in chain code as follows:

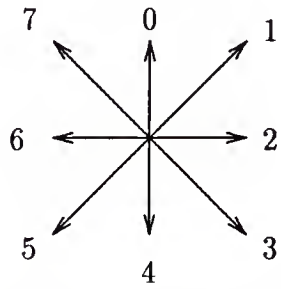
(1) Arbitrarily choose a pixel through which the curve passes. In the diagram, the start pixels for the chain code representation of objects 1 and 2 are marked by arrows.

(2) Traverse the curve in the clockwise direction. At each move from one pixel to the next, the direction of the move is recorded according to the convention shown in Figure 4.13(b). Objects 1 and 2 are represented by *1122102243244666666666* and *6666666611220022422424 46*, respectively. The alphabet is {0, 1, 2, 3, 4, 5, 6, 7} which is fixed and of constant size (8) and therefore satisfies the condition of Section 4.2. We may now use our visualization techniques of Chapter 3 to compare the two objects. For example, our methods would show that objects 1 and 2 share the segments S1 and S2 (Figure 4.13(c)) corresponding to *0224* and *24466666666661122*, respectively. Information on other common segments would also be available. The techniques of this paper make it possible to detect all patterns irrespective of the starting pixels chosen for the two objects.

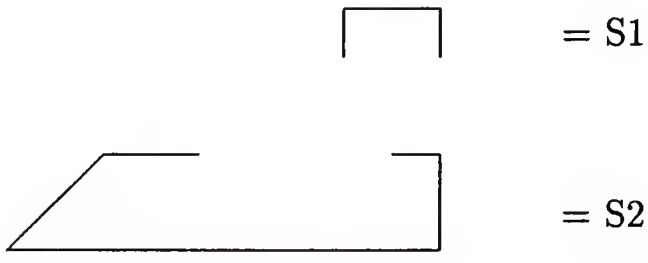
Circular strings may also be used to represent polygons in computer graphics and computational geometry [11]. Figure 4.14 shows a polygon which is represented by the following alternating sequence of lines and angles: b$\beta a\alpha e\alpha e\alpha c\beta c\beta e\beta a\alpha e\alpha e\alpha c\beta c\beta b\alpha c\alpha d\alpha c\alpha$, where α denotes a 90 degree angle and β, a 270 degree angle. The techniques of this paper would point out all instances of self similarity in the polygon, such as $a\alpha e\alpha e\alpha c\beta c$. Note, however, that for the methods to work efficiently, the number of lines and angles that are used to represent the polygons must be small and fixed.



Starting position for object 1 (a) Starting position for object 2



Chain code representations of directions
(b)



(c)

Figure 4.13. Representing closed curves by circular strings

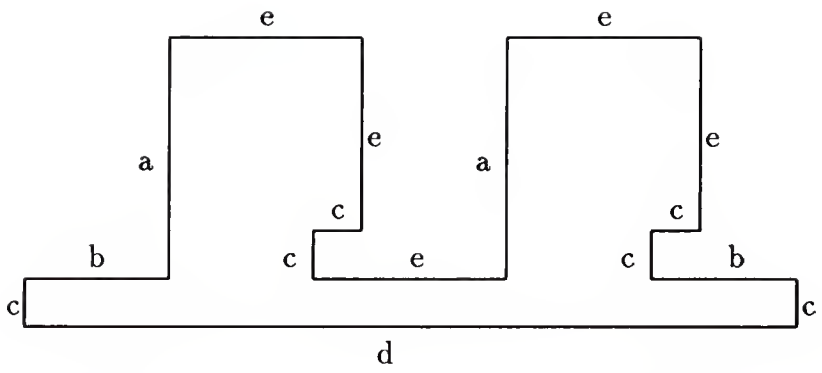


Figure 4.14. Representing polygons by circular strings

Closed curves defined using B-splines can be determined from their control polygons. If there exist two or more sufficiently long identical segments in the control polygon, then the curve fragments corresponding to those polygon segments would also be identical. Thus, similarity in closed curves represented by B-splines can also be detected by our techniques.

CHAPTER 5
EXTENSION TO BINARY TREES AND SERIES-PARALLEL GRAPHS

5.1 Tree Visualization

In this section, we consider the problem of tree visualization. Section 5.1.2 mentions some of its applications while Section 5.1.3 outlines the algorithms.

5.1.1 Problem Definition

We provide a specification for the problem of tree visualization:

1. Structure of Data to be Visualized: A binary tree BT of size n. Each node of the binary tree contains a key.

2. Structure of Patterns: A subtree of BT.

3. Maximality of Patterns: A subtree pattern is not maximal if its occurrences are all left or all right children of their parents and the subtrees rooted at their parents are all identical. Figure 5.1 shows an example tree BT of size 7. We will represent it using the *parenthesized infix* notation as *(((b)a(c))e((b)a(c)))* for the purposes of this paper. Here, the subtrees *(b)* and *(c)* are not maximal. However, *((b)a(c))* and *(((b)a(c))e((b)a(c)))* are maximal.

4. Measure of Similarity MS: If two patterns are identical, then **MS** = 1. Otherwise, **MS** = 0.

5. Display Model: A maximal subtree of BT is called a *displayable subtree* if it occurs at least twice in BT. All instances of the same displayable subtree are shaded in the same color. Different displayable subtrees are shaded in different

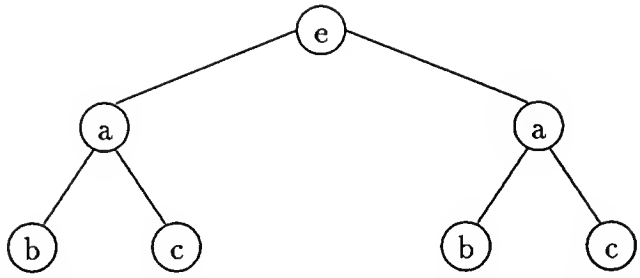


Figure 5.1. A labeled binary tree

colors. In the example of Figure 5.1, *((b)a(c))* is the only displayable subtree. So, BT would be displayed as shown in Figure 5.2.

We encounter the problem of subtree conflicts which is analogous to subword conflicts in strings. Consider the binary tree of Figure 5.3. The two displayable patterns are *(((b)a(c))d)* and *((b)a(c))*. Note that the latter is a subtree of the former. Consequently, the two subtrees cannot be shaded using different colors. Formally, a subtree conflict occurs between two displayable subtrees P_1 and P_2 iff one is a proper subtree of the other.

The problem of subtree conflicts may be solved by using an approach similar to that of Model 3 of Section 2.3. The resulting display of *(((((b)a(c))d)e(((b)a(c))d))f((b)a(c)))* is shown in Figure 5.4.

5.1.2 Applications

Binary trees are chiefly used as data structures in computer science [10]. Sophisticated debuggers attempt to display data structures at different points in the execution of a program [14, 15, 16]. Systems for algorithm animation also require displays of data structures [17, 18, 19]. The techniques of this section may be used in

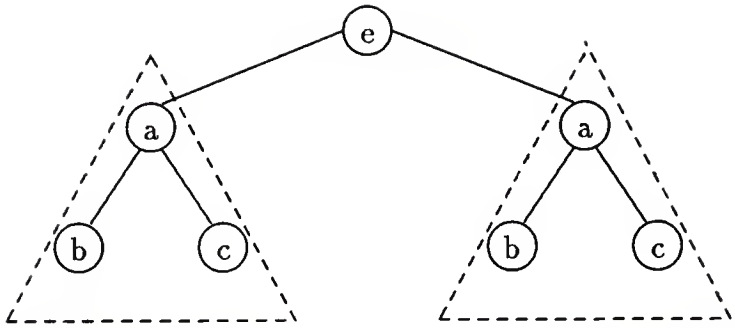


Figure 5.2. Highlighting displayable subtrees

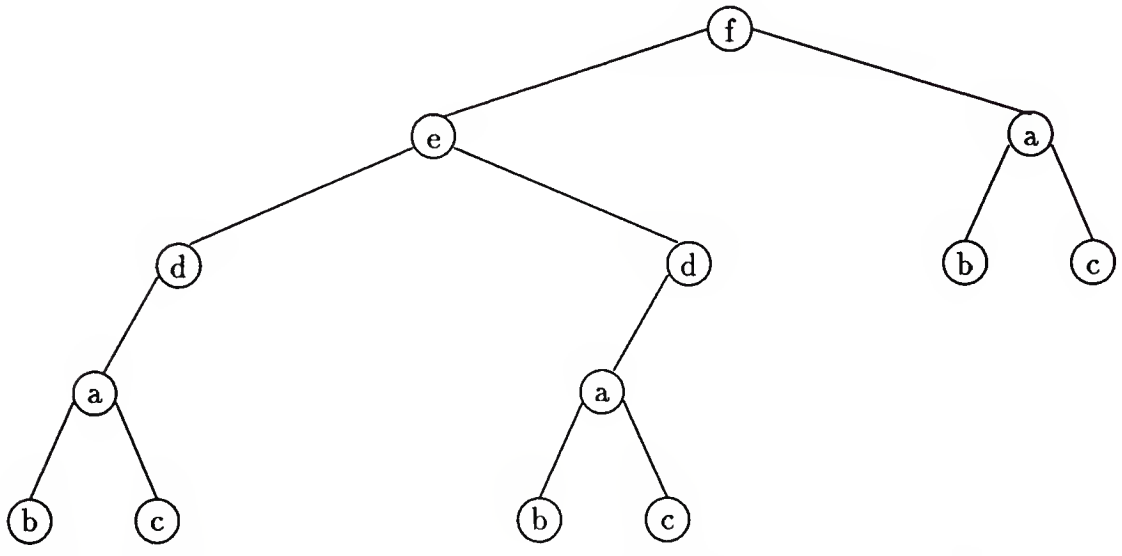


Figure 5.3. Subtree conflicts

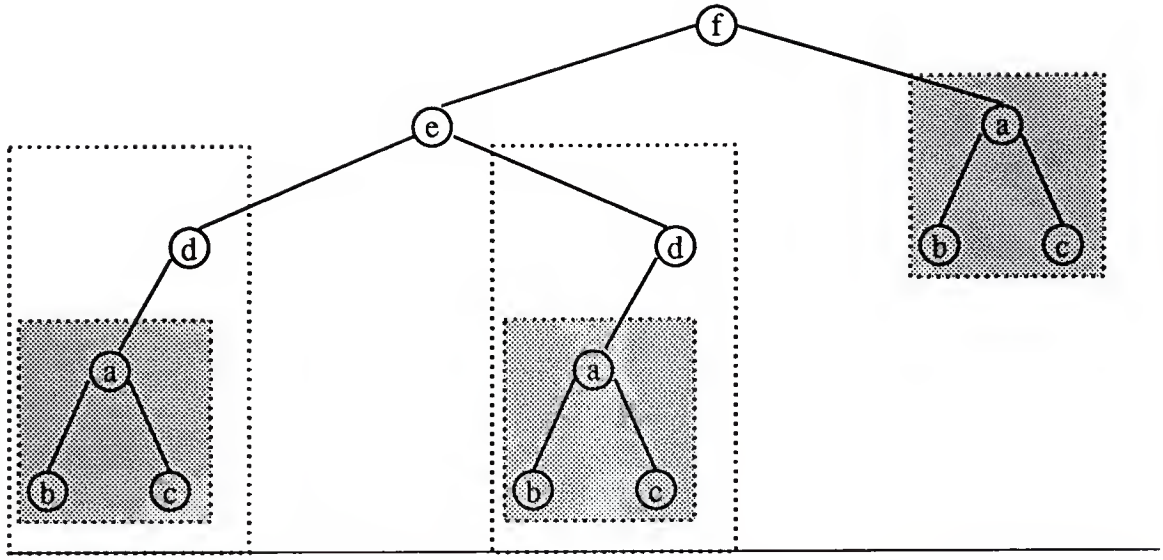


Figure 5.4. Displaying a tree with subtree conflicts

conjunction with tree layout algorithms [20, 21, 22] to provide a meaningful display of binary trees.

While the emphasis of this paper is on exploiting similarity to display discrete objects, we note that similarity detection is also useful in other applications.

For example, consider a forest of expression trees. Each expression tree corresponds to the computation of a variable. The leaf nodes of an expression tree correspond to operands while non leaf nodes correspond to operators (Figure 5.5). The techniques of this paper identify all common subexpressions in the expression trees. Specifically, common subexpressions which are embedded inside other common subexpressions are also detected.

Our techniques also provide information for the efficient storage of a forest of trees by converting it into a directed acyclic graph. For example, the tree of Figure 5.3 can be stored efficiently as in Figure 5.6.

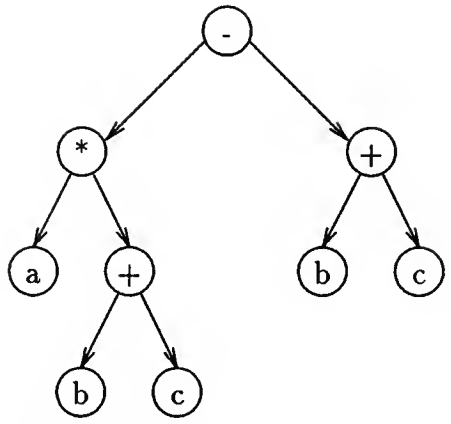


Figure 5.5. An expression tree

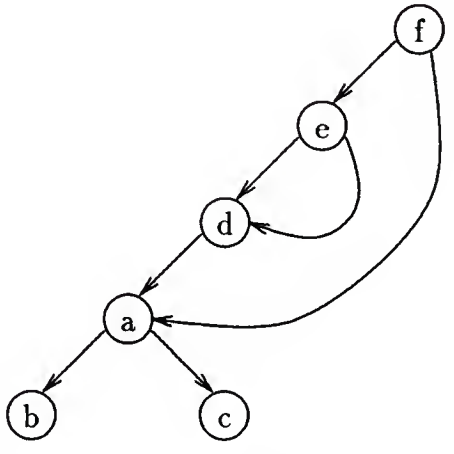


Figure 5.6. Tree compression by representation as a dag

5.1.3 Algorithms

Each node of the tree contains the fields: *lchild*, *rchild*, *key*, *id*, *leftid*, *rightid*, and *frequency*. The *id* and *frequency* fields of each node are initialized to 0. The *leftid* (*rightid*) field of a node is initialized to -1 if its *lchild* (*rchild*) field is *nil*; otherwise, it is initialized to 0.

Algorithm *IdentifySubtrees* of Figure 5.7 computes all displayable subtree patterns of BT. At the end of execution of *IdentifySubtrees*, all occurrences of a displayable subtree are assigned the same unique integer (which is stored in the *id* field of the root of each occurrence of the displayable subtree). The variable *NewId* is used as a counter.

StoreLeaves(BT,L) (line 1) traverses BT and stores pointers to its leaves in the linked list L. Each node in L contains two fields: *treenode*, which points to a node in BT and *next*, which points to the next element of L. M is a linked list containing nodes that are identical to those of L. Initially, M is the empty list (line 5). *Sort(L)* (line 6) sorts L on *(treenode↑.key, treenode↑.leftid, treenode↑.rightid)*. *Groups(L)* (line 7) identifies groups of nodes such that all nodes in a group contain identical values in *treenode↑.key*, *treenode↑.leftid*, and *treenode↑.rightid*. Each group is a contiguous sequence in L because of line 6. Let there be g groups L_k, for $1 \leq k \leq g$. L_k is a pointer to the first element in the k'th group. Let the number of nodes in L_k be s_k for $1 \leq k \leq g$. Note that a group represents all occurrences of a subtree pattern. The equality of the *leftid* and *rightid* fields guarantees that the left subtrees are identical and the right subtrees are identical. The equality of the *key* fields ensures that the roots are identical. In the first iteration, a group represents identical leaves.

Lines 8 and 9 ensure that all groups with at least two elements are considered. So, a group corresponding to a maximal subtree pattern also corresponds to a displayable

subtree pattern. All roots of the subtrees corresponding to a group are assigned a unique *id* (lines 11 and 22). Their *frequency* fields are also initialized (line 21) to the number of elements in the group, i.e. s_k. These will be used later to determine whether the subtree pattern corresponding to the group is maximal.

Lines 12 to 17 determine whether the subtree patterns rooted at either or both of the two children of $L_k \uparrow.treenode$ are displayable. If this is not the case, then the *id* fields of either the left children or right children or both, as appropriate, are reset (lines 35 and 36). Lines 23 to 34 initialize either the *leftid* or *rightid* field of the parent of the root of each subtree occurrence (depending on whether the particular root is a left child or a right child of its parent). If both, the *leftid* and the *rightid*, fields of the parent are non-zero, then a pointer to the parent is inserted into the linked list M (lines 27 and 33). Finally, the list L is deleted and the process is repeated with list M (lines 40 and 41).

Line 1 of *IdentifySubtrees* consumes $O(n)$ time, since it involves a traversal of BT. Let L_i denote the list L in the i'th iteration of the **while** statement of line 3, and $|L_i|$ denote the number of nodes in L_i ($1 \leq i \leq w$, where w represents the number of iterations of the **while** loop). Since each node in L_i corresponds to a distinct node in BT and since a node in BT is an element of at most one L_i, $\sum_{i=1}^{w} |L_i| \leq n$.

We now show that the i'th iteration of the **while** loop consumes $O(|L_i|)$ time. It is assumed that all keys are in the range $[1..n]$. Then, line 6 can be executed in $O(|L_i|)$ time, if radix sort is used to sort the linked list. The keys are relabeled so that they are in the range $[1..|L_i|]$. The relabeling can be done in $O(|L_i|)$ time as shown in Figures 5.8 and 5.9. Line 7 requires a pass over L_i and also consumes $O(|L_i|)$ time. Lines 8 to 39 essentially involve a pass over L_i, with $O(1)$ time being spent at each node. Line 40 also involves a pass over L_i and consumes $O(|L_i|)$ time. Thus, *IdentifySubtrees* consumes $O(n)$ time. (If the keys are not in the range $[1..n]$,

```
Algorithm IdentifySubtrees
1     StoreLeaves(BT,L)
2     NewId := 0;
3     while L is not empty do
4        begin
5           M := nil;
6           Sort(L);
7           Groups(L);
8           for k := 1 to g do {g is the number of groups obtained from line 7}
9              if s_k > 1 then {s_k is the number of nodes in group k}
10                begin
11                   NewId := NewId + 1;
12                   ResetLeft := false;
13                   ResetRight := false;
14                   if L_k ↑.treenode↑.lchild ≠ nil
15                   then if L_k ↑.treenode↑.lchild↑.frequency = s_k then ResetLeft := true;
16                   if L_k ↑.treenode↑.rchild ≠ nil
17                   then if L_k ↑.treenode↑.rchild↑.frequency = s_k then ResetRight := true;
18                   for j := 1 to s_k do
19                      begin
20                         current := L_k ↑.treenode;
21                         current↑.frequency := s_k;
22                         current↑.id := NewId;
23                         if current↑.parent↑.lchild :=current then
24                            begin
25                               current↑.parent↑.leftid := NewId;
26                               if current↑.parent↑.rightid ≠ 0
27                               then AddNode(M,current↑.parent);
28                            end
29                         else
30                            begin
31                               current↑.parent↑.rightid := NewId;
32                               if current↑.parent↑.leftid ≠ 0
33                               then AddNode(M,current↑.parent);
34                            end;
35                         if ResetLeft then current↑.lchild↑.id = 0;
36                         if ResetRight then current↑.rchild↑.id = 0;
37                         L_k := L_k ↑.next;
38                      end;
39                end;
40          DeleteList(L)
41          L := M;
42       end.
```

Figure 5.7. Algorithm for identifying displayable subtrees

{ An auxiliary array A and stack S, each of size n, are used. All elements of A are initialized to 0 at the beginning of the program. The stack is initially empty. A key k (guaranteed to lie between 1 and n) is relabeled by first checking $A[k]$ to determine whether it has already been assigned a label. If not, key k is assigned a label. Finally, the integer k is pushed onto S.}

```
Procedure Label(L)
begin
   count := 0;
   for each key k in L do
      if A[k] = 0 then
         begin
            count := count + 1;
            A[k] := count;
            Push(S,k);
         end;
end;
```

Figure 5.8. Procedure for relabeling keys

then they can be relabeled to satisfy this condition. However, the relabeling step would take $O(n\log k)$ time, where k is the number of distinct keys in BT.)

5.2 Geometric Series-Parallel Graph Visualization

In this section, we consider the problem of geometric series-parallel graph visualization. Section 5.2.2 mentions some of its applications while Section 5.2.3 outlines the algorithms.

5.2.1 Problem Definition

We provide a specification for the problem of series-parallel graph visualization:

1. Structure of Data to be Visualized: A geometric series-parallel graph SPG with n labeled vertices. Each vertex of SPG contains a character chosen from an alphabet Σ of constant size. "Fork" and "Join" vertices are not labeled.

{ The elements of array A are reset to 0 prior to the start of the next iteration of the **while** statement on line 3 of **Algorithm** *IdentifySubtrees*. }

```
Procedure ReInitialize;
begin
   while S is not empty do
      begin
         k := Pop(S);
         A[k] := 0;
      end
end
```

Figure 5.9. Procedure for re-initializing array A

Figure 5.10 shows a geometric series-parallel graph with 13 labeled vertices, 2 fork vertices, and 2 join vertices. We represent the series parallel graph as a string using the *double parenthesis* notation: *ab[(cd)(ef)(b)]ab[(cd)(ef)]*.

A geometric series-parallel graph differs from a regular series-parallel graph in that it describes the layout of a regular series-parallel graph in a plane. So, the graph obtained by exchanging (cd) and (ef) in Figure 5.10 would be a different geometric series-parallel graph from the one in Figure 5.10, even though both represent the same series-parallel graph.

2. Structure of Patterns: A geometric series-parallel subgraph of SPG.

3. Maximality of Patterns: A series-parallel subgraph pattern P_1 is not maximal iff all its instances are subgraphs of the same series-parallel subgraph P_2 and if these instances of P_1 all occur in the same relative geometric position in P_2.

4. Measure of Similarity **MS**: If two patterns are identical, then **MS** = 1. Otherwise, **MS** = 0.

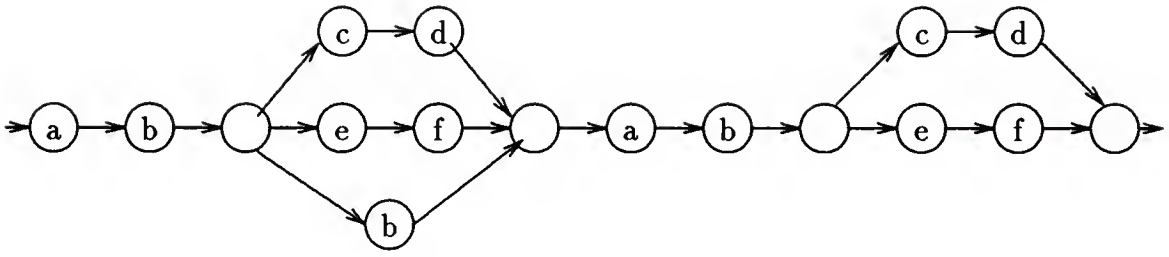


Figure 5.10. A geometric series-parallel graph

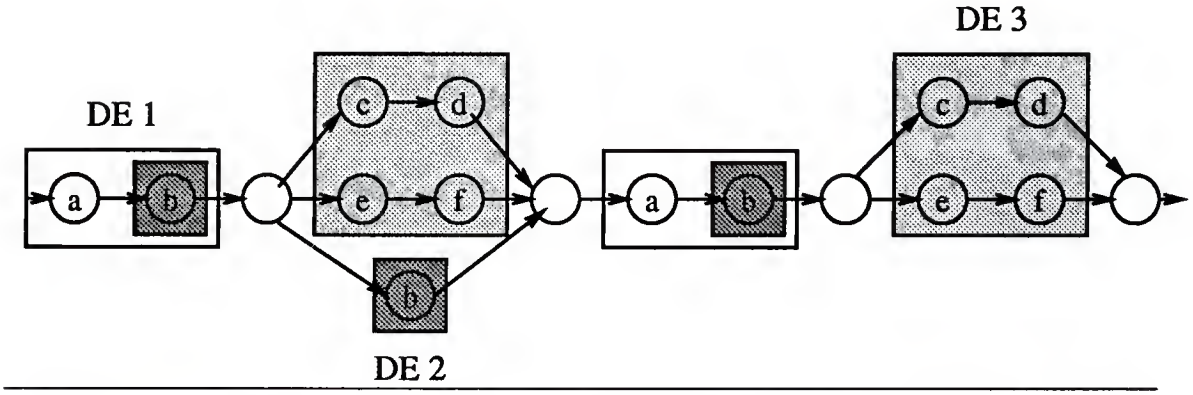


Figure 5.11. Highlighting displayable subgraphs

5. Display Model: A maximal subpattern of SPG is called a *displayable subgraph* if it occurs at least twice in SPG. All instances of the same displayable subgraph are shaded in the same color. Different displayable subgraphs are shaded in different colors. The displayable subgraphs are *ab*, *b*, and *(cd)(ef)*. So, the graph of Figure 5.10 may be displayed using a model similar to that of Model 3 of Section 2.3. as shown in Figure 5.11.

Algorithm *IdentifySubgraphs*

1. Convert geometric series-parallel graph SPG into a string S in the double parenthesis notation.
2. Compute $CSD(S)$
3. Extract displayable subgraphs of SPG from the displayable entities of $CSD(S)$. Each displayable subgraph is associated with the smallest displayable entity from which it was extracted.

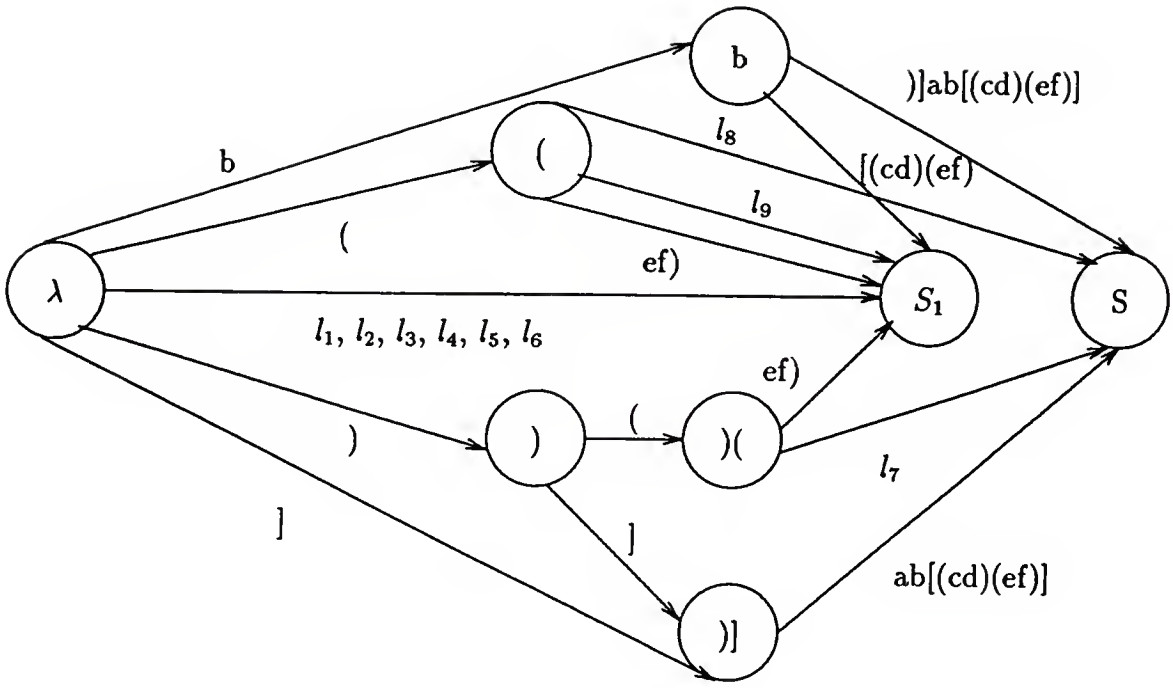
Figure 5.12. Computing displayable subgraphs

5.2.2 Applications

Series-parallel circuits are an important class of electronic circuits. These can be modeled by series-parallel graphs in which the vertices (other than fork and join vertices) represent active circuit elements. These vertices are labeled by the name/id of the active component they represent. The different branches of a fork/join are ordered top to bottom in some canonical way. For example, the lexical ordering of the double parenthesis string representation of the branches could be used. Common subgraphs represent common subcircuits. Circuit visualization/display systems are required to display circuits in such a way that common circuit substructures are easily identified.

5.2.3 Algorithms

Figure 5.12 outlines an algorithm for determining the displayable subgraphs of a geometric series-parallel graph. First, the geometric series parallel graph SPG is converted into a string S in the double parenthesis notation described earlier. Next, the csdawg $CSD(S)$ corresponding to S is constructed. Figure 5.13 shows $CSD(S)$ for $S = ab[(cd)(ef)(b)]ab[(cd)(ef)]$ (only re-edges are shown). Finally, the



l_1: ab[(cd)(ef)
l_2: [(cd)(ef)
l_3: cd)(ef)
l_4: d)(ef)
l_5: ef)
l_6: f)
l_7: b)]ab[(cd)(ef)]
l_8: b)]ab[(cd)(ef)]
l_9: cd)(ef)
S_1: ab[(cd)(ef)

Figure 5.13. Csdawg for graph of Figure 5.10, only re-edges are shown

displayable subgraphs of SPG are obtained from the displayable entities of S. This is achieved by extracting the longest substrings with no unmatched parenthesis from each displayable entity. The displayable subgraphs corresponding to the displayable entity *ab[(cd)(ef)* are *ab* and *(cd)(ef)* corresponding to DE 1 and DE 3 in Figure 5.11. Each displayable subgraph is associated with the vertex in $CSD(S)$ from which it was obtained so that all the locations of its occurrences in SPG can be retrieved. If a displayable subgraph is obtained from two or more displayable entities, then it is associated with the smallest displayable entity. Note that a unique smallest displayable entity must exist from the definition of an csdawg. Steps 1 and 2 of the algorithm each consume $O(n)$ time, while Step 3 consumes $O(n^2)$ time.

CHAPTER 6
SYSTEM INTEGRATION

6.1 Using the Object-Oriented Methodology

In this section we demonstrate that our algorithms are well-suited for implementation using the object-oriented paradigm. The *data abstraction* and *inheritance* features of the object-oriented paradigm are particularly applicable to our algorithms.

Most of our algorithms for the string, circular string, and series parallel graph discrete objects are essentially traversals or operations on the corresponding csdawg. High level operations on the csdawgs such as creating the csdawg, computing displayable entities, computing their locations, computing display conflicts, etc make use of lower level operations on the csdawgs such as finding a vertex in the csdawg corresponding to a substring (subgraph), modifying the contents of a vertex, etc. Thus, the concept of data abstraction which views data as a black box whose contents can only be accessed through its operations can be applied naturally to csdawgs.

Many operations are implemented identically for each of the three csdawgs. These are usually low level operations such as locating the vertex corresponding to a string, or modifying the contents of a vertex, etc. Other operations (usually higher level operations), while not implemented identically for the three discrete objects, are similar. Typically, a high level operation on a circular string or a series parallel graph can be implemented by the corresponding operation on a linear string preceded by a simple preprocessing step and followed by a simple postprocessing step.

We create a class for each csdawg corresponding to the three discrete objects: *LinearCsdawg, CircularCsdawg, SeriesParallelCsdawg. LinearCsdawg* is a base class,

while *CircularCsdawg* and *SeriesParallelCsdawg* are classes derived from *LinearCsdawg*. The operations that are implemented identically for each class are implemented *once* as operations on *LinearCsdawg*. These operations are not implemented separately for *CircularCsdawg* or *SeriesParallelCsdawg* as they can be derived from those for *LinearCsdawg*. Other operations are implemented separately for each class because they are implemented differently. The operations on *CircularCsdawg* and *SeriesParallelCsdawg* may invoke the corresponding operations on *LinearCsdawg*.

6.2 Overview of a visualization system

In this section, we provide an overview of a visualization system which uses the similarity paradigm. Figures 6.1 and 6.2 show flowcharts which outline the flow of control of such a system and indicate the algorithms used at each stage. The rectangular boxes denote computations, the diamond structured boxes denote decision points. Ellipses denote algorithms and routines. The broken lines connect computational units to the routines and algorithms which they invoke. The solid lines denote the flow of control. The broken line boxes of Figure 6.2 labeled I and A denote the entire flow sequences corresponding to the interactive and automatic options in Figure 6.1, respectively.

First, the discrete object (or objects) is input to the system. If the object is a string, circular string, or a geometric series parallel graph, then the corresponding csdawg is constructed. If the object is a binary tree, then *IdentifySubtrees* is executed. At this stage, all displayable sub objects and their locations in the object are obtained.

Next, the user chooses the mode in which he/she wishes to operate the system.

In the interactive mode, the user chooses the displayable entities he/she wishes to have highlighted. This is an incremental process in which the user makes local queries about the length of a particular occurrence, the number and nature of conflicts with

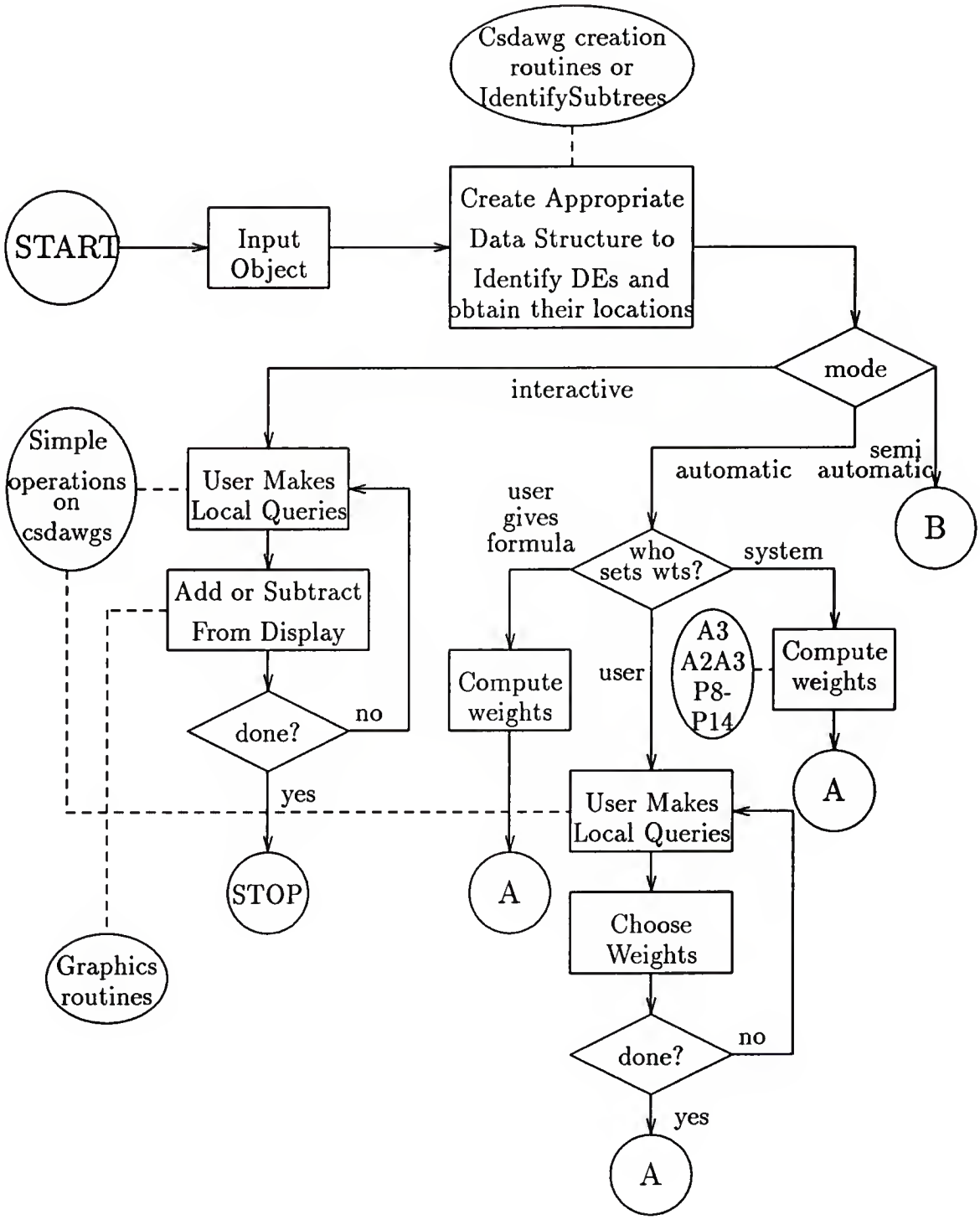


Figure 6.1. Flow of control diagram for a system

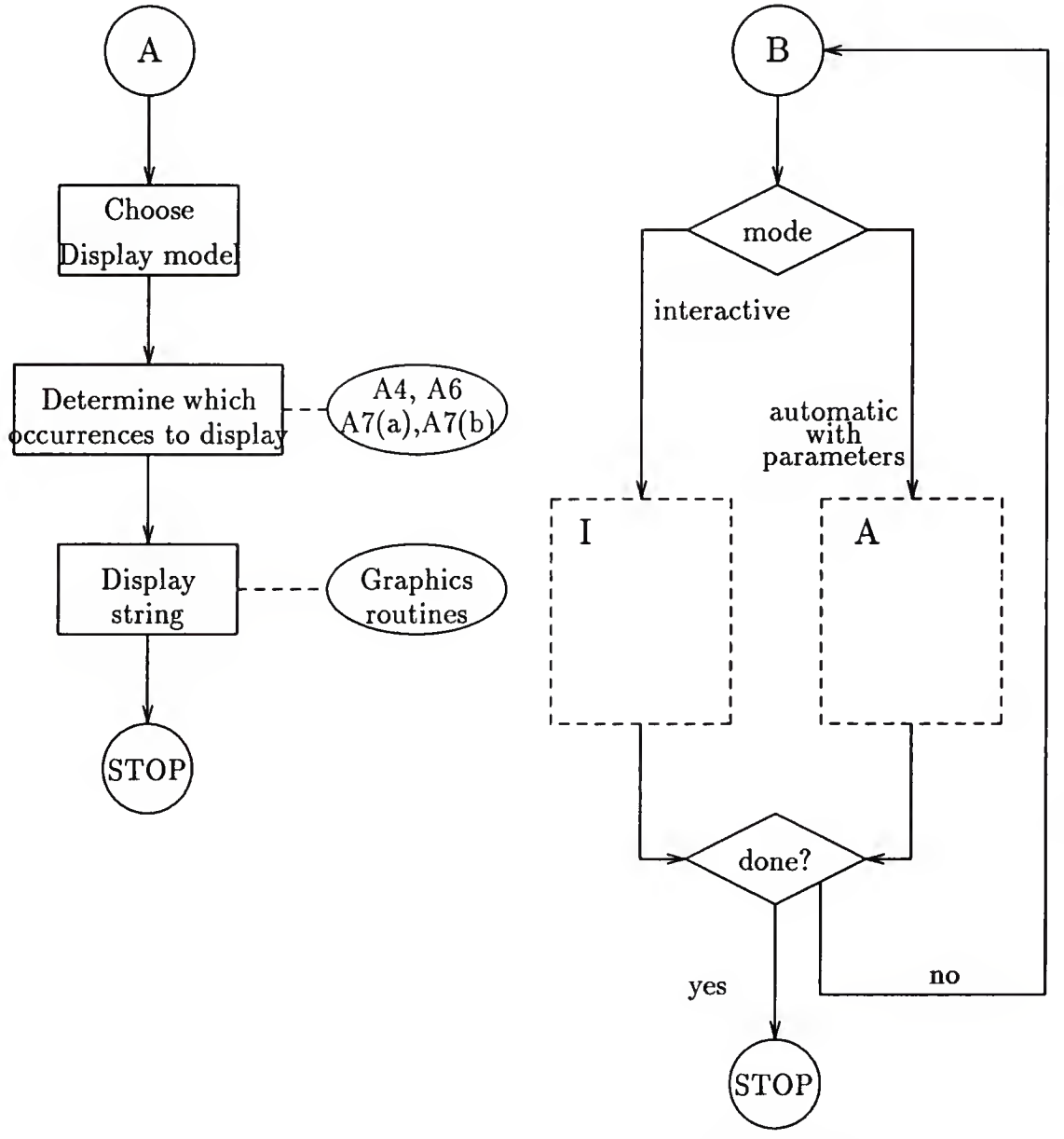


Figure 6.2. Flow of control diagram for a system (contd.)

it, etc. These queries would typically involve simple operations on the csdawg. The user would then, depending on the outcome of these queries, decide whether he/she wishes to have the particular occurrence highlighted or not, or whether he/she no longer wishes to highlight another occurrence that was previously highlighted. This cycle would be repeated until the user was satisfied with the display. In the automatic mode, numeric weights are assigned to each occurrence. The user decides whether he/she wishes to assign weights to the occurrences him/her self or whether to let the system do it. There are two ways for the user to assign weights to the occurrences. The user could choose a weight for each occurrence. Or he/she could specify a formula for the computation of the weights of the occurrences. This formula could take the form of a simple arithmetic expression which uses attributes of the occurrences such as length, frequency, etc or it could take the form of a complex computational procedure with decision making components. Alternatively, the user could let the system compute the weights using one of a number of default formulae which are maintained in a library. The automatic mode would make use of exhaustive global queries such as those supported by algorithms *A3, A2A3*, or more selective queries such as **P8-P14.**

The semi-automatic mode essentially involves user-controlled switching between the automatic and interactive modes. We expect that the semi-automatic mode will be the most widely used mode.

Once the weights are determined, the user is prompted for the display model that he/she wishes to use. The display model could be one of those provided by the system or one that is user-defined. The system then chooses the occurrences that are to be highlighted under the particular display model. This is supported by algorithms *A4, A6, A7(a)*, and *A7(b).*

A facility would be provided to store the contents of an interactive session so that the user can continue from where he/she left off at a later time.

CHAPTER 7
CONCLUSIONS

In this dissertation, we have proposed a model for the visualization of labeled discrete objects. This model was based on the similarity paradigm. We studied the problem of visualization conflicts that arose as a consequence of the similarity paradigm and provided methods to overcome them.

We identified a number of algorithmic questions that would have to be answered to implement the visualization model. We provided theoretically efficient algorithms to implement the visualization model for linear strings. Most of our algorithms utilized the csdawg data structure for linear strings. We implemented several of these algorithms to ensure that they were practically viable. These algorithms were implemented in C++, making use of object-oriented mechanisms provided in that language.

Next, we defined the csdawg for circular strings, which may be used to extend our algorithms for linear strings to circular strings. We expect that it can also be used for other string matching applications involving circular strings. An important feature of the csdawg for circular strings is that it is easy to implement and use when corresponding techniques for csdawgs for linear strings are already available. While we defined the csdawg for a single circular string, we note that it can be easily extended to represent a set containing both linear and circular strings.

We also applied our techniques to the problems of binary tree and geometric series-parallel visualization.

We now list some possible directions for future research in this area.

1. Extension to approximate similarity- Throughout this dissertation, we assumed a simple definition of "measure of similarity;" i.e., if two subpatterns are identical, then the measure of similarity is 1. Otherwise, it is 0.

   Any extension to approximate similarity would require a definition of distance between two patterns. A possibility is the Levenshtein distance, which is defined as the smallest number of substitutions, insertions, and deletions required to change one pattern into the other.

   Once a definition for distance has been chosen, we are confronted with the issue of displaying similar patterns. First, we will be required to choose a *distance threshold* (dt) such that if the distance between two patterns is greater than dt, then it is not necessary to display the patterns so that they look similar. If the distance between two patterns is less than dt, then we are required to have similar displays for them. In this case, the smaller the distance the greater the similarity in the displays. One way of achieving this is to use similar colors to represent similar patterns. This would involve investigating issues in color theory and human perception.

2. Extension to other discrete objects- In this dissertation, we have studied the visualization problem for linear and circular strings, binary trees, and series-parallel graphs. A possible direction for future research would be to extend the results to other, more complex discrete objects such as 2d-strings, general graphs, etc.

3. Dynamic discrete objects- The discrete objects considered in the dissertation are static; i.e., they do not change with time. Another possible direction for future research would be to extend our visualization model to discrete objects which change with time.

A first step in this direction would be to select a set of operations on the discrete object that we wish to support. For example, in a linear string these operations might be insertion, deletion, and substitution. Once these operations have been developed, models which indicate how the display is to be updated under each of these operations would have to be developed.

REFERENCES

[1] J. Beddow. Shape Coding of Multidimensional Data on a Microcomputer. In *Proceedings of the First IEEE Conference on Visualization*, pages 238–246, San Francisco, 1990.

[2] G.M. Morris . The Matching of Protein Sequences using Color Intrasequence Homology Displays. *J. Mol. Graphics*, 6:135–142, 1988.

[3] D. Sankoff, J.B. Kruskal. *Time Warps, String Edits, and Macromolecules: the Theory and Practice of Sequence Comparison.* Addison-Wesley, Reading, Massachussetts, 1983.

[4] B. Clift, D. Haussler, T.D. Schneider, and G.D. Stormo . Sequence Landscapes. *Nucleic Acids Research*, 14(1):141–158, 1986.

[5] Mila E. Majster and Angelika Reiser. Efficient on-line construction and correction of position trees. *SIAM Journal on Computing*, 9(4):785–807, November 1980.

[6] Edward McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, April 1976.

[7] A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M.T. Chen, J. Seiferas. The Smallest Automaton Recognizing the Subwords of a Text. *Theoretical Computer Science*, (40):31–55, 1985.

[8] M. T. Chen and Joel Seiferas. Efficient and elegant subword tree construction. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words*, NATO ASI Series, Vol. F12, pages 97–107. Springer-Verlag, Berlin Heidelberg, 1985.

[9] A. Blumer, J. Blumer, D. Haussler, R. McConnell, and A. Ehrenfeucht. Complete Inverted Files for Efficient Text Retrieval and Analysis. *J. ACM*, 34(3):578–595, 1987.

[10] E.Horowitz, S. Sahni. *Fundamentals of Data Structures in Pascal, 3rd Edition.* Computer Science Press, New York, 1990.

[11] S.L. Tanimoto. A method for detecting structure in polygons. *Pattern Recognition*, 13(6):389–394, 1981.

[12] R. Gonzalez, P. Wintz. *Digital Image Processing, 2nd Edition.* Addison Wesley, Reading, Massachussetts, 1987.

[13] D.P. Mehta and S. Sahni. Computing display conflicts in string visualization. Technical Report 91-024, University of Florida, 1991.

[14] Brad A. Myers. Incense: A System for Displaying Data Structures. *Computer Graphics*, 17(3):115–125, 1983.

[15] G.P. Brown, R.T. Carling, C.F. Herot, D.A. Kramlich, P. Souza. Program Visualization: Graphical Support for Software Development. *IEEE Computer*, 18(8):27–35, 1985.

[16] David B. Baskerville. Graphic presentation of data structures in the dbx debugger. Technical Report UCB/CSD 86/260, University of California, Berkeley, 1985.

[17] Marc H. Brown and Robert Sedgewick. A System for Algorithm Animation. *Computer Graphics*, 18(3):177–186, 1984.

[18] Marc H. Brown and Robert Sedgewick. Techniques for Algorithm Animation. *IEEE Software*, 2(1):28–39, 1985.

[19] Marc H. Brown. *Algorithm Animation.* PhD dissertation, Brown University, Department of Computer Science, 1987.

[20] Charles Wetherell and Alfred Shannon. Tidy Drawings of Trees. *IEEE Transactions on Software Engineering*, (5):514–520, 1979.

[21] Edward M. Reingold and John S. Tilford. Tidier Drawings of Trees. *IEEE Transactions on Software Engineering*, (2):223–228, 1981.

[22] Jean G. Vaucher. Pretty-Printing of Trees. *Software-Practice and Experience*, 10:553–561, 1980.

[23] D.P. Mehta and S. Sahni. A data structure for circular string analysis and visualization. Technical Report 91-025, University of Florida, 1991.

[24] D.P. Mehta and S. Sahni. Models and techniques for the visualization of labeled discrete objects. Technical Report 91-032, University of Florida, 1991.

[25] D.P. Mehta and S. Sahni. Models and Techniques for the Visualization of Labeled Discrete Objects. In *Proceedings of the 1992 ACM/SIGAPP Symposium on Applied Computing*, pages 1224–1233, Kansas City, 1992.

[26] D.P. Mehta and S. Sahni. Computing display conflicts in string and circular string visualization. In *Proceedings of the Third Symposium on Combinatorial Pattern Matching*, pages 241–258, Tucson, 1992.

BIOGRAPHICAL SKETCH

Dinesh Mehta was born on December 31, 1965, in New Delhi, India. He received his undergraduate degree in computer science and engineering from the Indian Institute of Technology at Bombay in July 1987 and his masters in computer and information sciences at the University of Minnesota at Minneapolis in February 1990. He will receive his Doctor of Philosophy degree in computer and information sciences at the University of Florida at Gainesville in August 1992. His research interests include visualization, user interfaces, parallel computing, VLSI CAD, and data structures and algorithms.

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Sartaj Sahni, Chair
Professor of Computer and
Information Sciences

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Gerhard Ritter
Professor of Computer and
Information Sciences

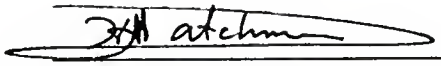
I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

John Staudhammer
Professor of Computer and
Information Sciences

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Theodore Johnson
Assistant Professor of Computer and
Information Sciences

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.



Haniph Latchman
Assistant Professor of
Electrical Engineering

This dissertation was submitted to the Graduate Faculty of the College of Engineering and to the Graduate School and was accepted as partial fulfillment of the requirements for the degree of Doctor of Philosophy.

August 1992.

Winfred M. Phillips
Dean, College of Engineering

Madelyn M. Lockhart
Dean, Graduate School

UNIVERSITY OF FLORIDA
3 1262 08285 435 6